Under consideration for publication in Math. Struct. in Comp. Science

A Type System for Discretionary Access Control

MICHELE BUGLIESI[1], DARIO COLAZZO[2],

SILVIA CRAFA[3], and DAMIANO MACEDONIO[1] †

[1] *Dipartimento di Informatica, Università Ca' Foscari, Venice.*
[2] *LRI, Université Paris Sud.*
[3] *Dipartimento di Matematica Pura e Applicata, Università di Padova.*

Received

Discretionary Access Control (DAC) systems provide powerful resource management mechanisms based on the selective distribution of capabilities to selected classes of principals. We study a type-based theory of DAC models for a process calculus that extends Cardelli, Ghelli and Gordon's pi-calculus with groups (Cardelli et al., 2005). In our theory, groups play the rôle of principals, the unit of abstraction for our access control policies, and types allow the specification of fine-grained access control policies to govern the transmission of names, to bound the (iterated) re-transmission of capabilities, to predicate their use on the inability to pass them to third parties. The type system relies on subtyping to achieve a selective distribution of capabilities, based on the groups that control the communication channels. We show that the typing and subtyping relationships of the calculus are decidable. We also prove a type safety result, showing that in well-typed processes *(i)* all names flow according to the access control policy specified by their types, and *(ii)* are received at the intended sites with the intended capabilities. We illustrate the expressive power and the flexibility of the typing system on several examples.

1. Introduction

Type systems have been applied widely in process calculi to provide static guarantees for a variety of safety and security properties, including access control (Hennessy and Riely, 2002a; Chothia et al., 2003), non-interference (Kobayashi, 2005; Pottier, 2002; Honda et al., 2000), and secrecy (Abadi and Gordon, 1997; Cardelli et al., 2005).

The focus of the present paper is on Discretionary Access Control models (Lampson, 1974; Samarati and di Vimercati, 2001), i.e., models that support powerful resource management policies based on fine-grained mechanisms to control the use and the distribution of capabilities. To motivate our approach, we start with a long-known example on types for the pi-calculus (Pierce and Sangiorgi, 1996):

$$S = !s(x).\overline{print}\langle x\rangle \qquad\qquad C = \overline{s}\langle j_1\rangle.\overline{s}\langle j_2\rangle\ldots$$

S represents a printer spooler, serving requests from a channel s, while C specifies a client sending

† Work partially supported by M.I.U.R. (Italian Ministry of Education, University and Research) under contract n. 2005015785, and Università di Padova under contract n. CPDA061331.

jobs $j_1, j_2, \ldots$ to the spooler. A property we may wish to show of a system in which S and C run in parallel is that all the jobs sent by C are eventually received and printed by S. However, while such guarantees can be made for the system $S \mid C$ in isolation, they may hardly be enforced in more general situations: in fact, a misbehaved client such as $C' = s(x).s(y)\ldots$ may legally participate in the protocol to steal C's jobs, as in $S \mid C \mid C'$. The type system developed in (Pierce and Sangiorgi, 1996) prevents this undesired behaviour by relying on capability types to control the use of channels in well-typed processes. Applied to our example, that typing system would rule out the misbehaved client by ensuring that clients only receive write capabilities on the channel s, and by reserving the read capability on s just to the spooler:

$$c(y : (T)^{\mathsf{w}}).C) \;\mid\; (\nu s : (T)^{\mathsf{rw}})\overline{c}\langle s\rangle.S$$

The types $(T)^{\nu}$ are the types of channels with payload of type T and capabilities for reading, writing, or both, depending on whether ν is r, w, or rw, respectively. By delivering the channel s at the type $(T)^{\mathsf{w}}$ on the channel c shared with its clients, the spooler enforces a policy whereby (well-typed) clients may not intercept any job sent on channel s.

While effective in controlling the transfer of access rights between the two partners of a synchronization (S and C in the example), the system of capability types of (Pierce and Sangiorgi, 1996) provides limited control on the way that resources and their associated access rights are propagated to third parties. For instance, a client may wish to prevent its jobs from being read by processes other than the spooler S, and hence to disallow situations like the following, where the spooler forwards the jobs it receives to process SPY:

$$!s(x).\overline{log}\langle x\rangle.\overline{print}\langle x\rangle \;\mid\; \overline{s}\langle j_1\rangle.\overline{s}\langle j_2\rangle \;\mid\; log(y).SPY$$

The capability-based access control from (Pierce and Sangiorgi, 1996) is of little help here, unless one resorts to a more complex coding of the system, or else imposes overly restrictive conditions (e.g. prevent the spooler from writing on all public channels). A similar problem arises in the following variation of the protocol, in which the client requests a notification that his jobs have been printed:

$$S = (\nu print)\ (!s(x).\overline{print}\langle x\rangle \mid !print(x).(P \mid \overline{ack}\langle x\rangle)) \qquad C = \overline{s}\langle j_1\rangle.ack(x).\overline{s}\langle j_2\rangle.ack(y)$$

As in the previous case, a standard capability-based type system will fail to detect violations of the intended protocol by malicious (or erroneous) spoolers that discard jobs by, say, running the process $s(x).\overline{ack}\langle x\rangle$.

To counter these problems, we propose a novel typing system that complements the access control mechanisms based on capability types with stronger support for controlling the flow of resources among the components of a system. Our approach draws on putting more structure in the syntax of types so as to allow the specification of richer access control policies. In the new typing system, the types take the form $\mathsf{G}[T \parallel \Delta]$, where G is a group (or principal) name, T a capability type à la (Pierce and Sangiorgi, 1996) and Δ a delivery policy. If v is a value of type $\mathsf{G}[T \parallel \Delta]$, then G represents the authority in control of v, T describes the structure of v, and Δ specifies the legal flows of v along the channels of the system. To illustrate, in our running example, the type

$$\mathsf{JOB} = \mathsf{Client}[\ \mathsf{data} \parallel \mathsf{Spooler} : \mathsf{data} \rightarrow \mathsf{Printer} : \mathsf{data} \rightarrow \mathsf{Client} : \mathsf{data}\ ]$$

specifies a class of jobs controlled by clients, whose data (e.g., file descriptors) may only be delivered to the spooler, then passed on to the printer, and only then re-transmitted back to clients for notification. Similarly, the type

$$\mathsf{JOB} \downarrow \mathsf{Spooler} = \mathsf{Client}[\ \mathsf{data}\ \|\ \mathsf{Printer} : \mathsf{data} \to \mathsf{Client} : \mathsf{data}\ ]$$

identifies jobs that have "flown one hop" (to the spooler) along their intended delivery path, reaching the spooler.

Based on the delivery policy specified by type JOB, our static typing system will make the following guarantees for any composition of well-typed processes: *(i)* no notification is given to clients for jobs that have not previously been sent to a printer, and *(ii)* no job is received by the printer unless it has first been delivered to the spooler. In particular, assuming $j : \mathsf{JOB}$ and $s : \mathsf{Spooler}[(\mathsf{JOB} \downarrow \mathsf{Spooler})^{\mathsf{w}}\ \|\ \Delta]$, our type system allows the transmission of j over s, and guarantees that j will flow as specified in $\mathsf{JOB} \downarrow \mathsf{Spooler}$, once received on s. Thus, these typing assumptions make the spooler $!s(x).\overline{print}\langle x\rangle$ well-typed, as desired. On the other hand, they are also effective in ruling out the malicious spooler $s(x).\overline{ack}\langle x\rangle$ as ill-typed because it fails to comply with the policy associated with channel s.

By combining our delivery policies with the traditional access control based on subtyping à la (Pierce and Sangiorgi, 1996), our type system makes it possible to selectively advertise values at different types depending on the groups to which they are delivered. To illustrate, we may refine our printer example by stating $s : \mathsf{SPOOL}$ where

$$\mathsf{SPOOL} = \mathsf{Spooler}[\ (\mathsf{JOB} \downarrow \mathsf{Spooler})^{\mathsf{rw}}\ \|\ \mathsf{Client} : (\mathsf{JOB} \downarrow \mathsf{Spooler})^{\mathsf{w}}\,]$$

demands that when s is delivered to the client, it be received at a type that only provides write access so that no well-typed client may intercept jobs directed to the spooler. Now, assuming $c : \mathsf{Client}[\ (\mathsf{SPOOL} \downarrow \mathsf{Client})^{\mathsf{rw}}\ \|\ \Delta]$, we achieve the desired security guarantees of the spooling system by structuring it as shown below:

$$c(y : \mathsf{SPOOL} \downarrow \mathsf{Client}).C \ \mid\ (\nu s : \mathsf{SPOOL})\overline{c}\langle s\rangle.S$$

Here $\mathsf{SPOOL} \downarrow \mathsf{Client} = \mathsf{Spooler}[\ (\mathsf{JOB} \downarrow \mathsf{Spooler})^{\mathsf{w}}]$ is the type assigned to $s : \mathsf{SPOOL}$ once s has reached the client (this type has an empty delivery policy, because values of type SPOOL should not be further re-transmitted after having been delivered to a client).

We formalize our approach with a typed extension of the pi-calculus with groups from (Cardelli et al., 2005). The new calculus is structured in two layers. At the lower level, we find a process calculus that inherits the syntax of the pi-calculus with groups and extends its typing and subtyping systems with a richer class of resource and capability types to capture fine-tuned access control policies such as those discussed earlier. The static typing system ensures the compile-time detection of violations of such policies: in particular, a type preservation theorem allows us to derive a safety result stating that all well-typed processes comply with the discretionary access control policies governing the use of resources. We show that subtyping is decidable, and that so is the type checking problem.

On top of the process calculus, we then introduce our notion of principals by interpreting the groups from the underlying processes as principal names, and construe the latter as the unit of abstraction for access control. We illustrate the flexibility of the type system by showing how it supports DAC security models based on decentralized policies where the owner of an object can delegate to others the right to specify authorizations, possibly with the ability of further delegating it.

1.1. *Related work*

While we are not aware of any approach specifically targeted at access control mechanisms like the ones we have outlined, our work is clearly related to a large body of literature on type-based security in process calculi. Several type system encompass various forms of access control policies in distributed systems. Among them, (Chothia et al., 2003) proposes a form of distributed access control based on typed cryptographic operations; the work on the Dπ calculus (Riely and Hennessy, 1998), has produced fairly sophisticated type systems (Hennessy and Riely, 2002b; Hennessy et al., 2005) to control the access to the resources advertised at the different locations of a distributed system. None of these systems, however, addresses the kind of discretionary policies we consider here. More precisely, in Dπ, resources are created at a specific, unique, type and then delivered to different parties at different (super)types on different channels. Unlike in our system, however, the 'delivery policy' of a value is not described (nor prescribed) by its type. As a consequence, the only guarantee offered by the type system is that names delivered at type T will be received, and re-transmitted freely, at super-types of T. As we will show, our types may be employed to specify, and enforce, much more expressive policies. Other related work on access control in distributed systems has been carried in the context of the language KLAIM (Nicola et al., 2000) and its extensions to μKLAIM (Gorla and Pugliese, 2003) with type systems that enable the dynamic exchange of access rights targeted at access control mechanisms like the ones presented in the present paper, also based on a notion of *region* which in some respects is akin to our idea of group (Nicola et al., 2006).

Type systems have also been proposed to control implicit information flows determined by the behaviour of system components (see (Kobayashi, 2005; Pottier, 2002; Honda et al., 2000; Sewell and Vitek, 2003) among others). These type systems trace the causality relations between computational steps in order to detect covert channels. We follow a different approach to express and verify the delivery of (and the access to) the system resources.

Our approach is also related to the large body of existing work on security automata. In fact, the delivery policies we express in our type system could equivalently be described as finite-state automata whose states are structural types and edges are labelled with groups. On the other hand, security automata have traditionally been employed to provide for run-time system monitoring (Schneider, 2000) rather than as a basis for the development of static, type-based, security analysis.

Session types (Honda et al., 1998) have originally been proposed with motivations similar to ours to express and gain control over the different steps of communication protocols. Recently, they have been used extensively to structure the interaction among the components of a distributed system, cf. (Carbone et al., 2007; Honda et al., 2008). At the core of session types, lie mechanisms to regulate the sequence of events occurring in an interaction session. Specifically, a session type prescribes the sequence of states for a channel during a session, where each state is encoded as the type of values that may legally be exchanged over the channel at that stage. Instead, our resource types, encode the states of the values exchanged. As such, besides being technically different, session types and resource types target dual objectives: session types help structure the interaction between parties, while resource types help trace and control the flow of information among those parties.

1.2. *Plan of the paper*

Section 2 illustrates the syntax and the operational semantics of the process calculus. Section 3 introduces the notion of principals and illustrates its use with several examples. Section 4 describes the typing system. Section 5 details the proof of type safety and re-establishes the secrecy result of the pi-calculus with groups of (Cardelli et al., 2005). Section 6 proves that typing and subtyping are decidable. Section 7 concludes the presentation with final remarks.

A preliminary version of the present paper appeared as (Bugliesi et al., 2004).

2. A pi-calculus with groups and delivery types

We start by introducing the process calculus that constitutes the core of our formalization. The calculus is derived from the process calculus with groups defined in (Cardelli et al., 2005), which we henceforth refer to as πG. The syntax is defined by the following productions:

$$P, Q \ ::= \ \mathbf{0} \ \mid \ c(x_1 : \tau_1, \ldots, x_h : \tau_h).P \ \mid \ \overline{c}\langle a_1, \ldots, a_h \rangle.P \ \mid \ (\nu n : \tau)P \ \mid \ (\nu \mathsf{G})P \ \mid \ P|Q \ \mid \ !P$$

We presuppose countable sets of names $m, n, \ldots, s, t$, basic values bv, and variables $x, y, .., z$. We reserve a, b, c to range over all sets, with the proviso that basic values may not occur as channel names in the input/output prefixes of a process: this condition is enforced by the typing system, in Section 4. We also presuppose a countable set of *group names* $\mathsf{G}_1, \mathsf{G}_2, \ldots$, and reserve '$\mathsf{any}$' as a distinguished group symbol. As usual, a restriction $(\nu n{:}\tau)P$ binds the name n, and an input $c(x_1 : \tau_1, \ldots, x_h : \tau_h).P$ binds the variables $x_1, \ldots, x_h$: in both cases the scope of the binder is P. Similarly, a group creation $(\nu \mathsf{G})P$ binds the group name G in P. The notions of free names, variables and groups, noted $fn(P)$ and $fv(P)$ and $fg(P)$ respectively, arise as expected. The distinguished group 'any' may not occur in a group creation, hence it is always free.

As in (Cardelli et al., 2005), groups provide our process calculus with a mechanism for privacy. Briefly, one may define a name of group G, and then confine it within the scope of G, as in $(\nu n : \mathsf{G})P$. Although group creation is dynamic, group names cannot be communicated: hence no name belonging to a secret group G can be leaked outside the scope of G's original declaration. Drawing on the work of (Myers and Liskov, 2000), we extend these ideas and structure our types so that they convey information on how names may legally be propagated among (the channels of) the groups in a system.

The syntax of types is defined by the following productions:

$$\begin{array}{rll} \textit{Resource Types} & \tau \ ::= \ \mathsf{G}[T \parallel \Delta] & (\mathsf{G} \neq \mathsf{any}, \Delta \ \textit{closed}) \\ \textit{Structural Types} & T \ ::= \ \mathsf{B} \mid (\tau_1, \ldots, \tau_n)^{\nu} & (\nu \in \{\mathsf{r}, \mathsf{w}, \mathsf{rw}\}) \\ \textit{Delivery Policies} & \Delta \ ::= \ \varepsilon \mid \sum_i \mathsf{G}_i : T_i \to \Delta_i \mid X \mid \mu X.\Delta & (i \neq j \Rightarrow \mathsf{G}_i \neq \mathsf{G}_j) \end{array}$$

Resource types specify the group name, the structural type and the delivery policy of each value.

The role of the group name is twofold. Like in the calculus πG, the presence of G in the typing $n : \mathsf{G}[T \parallel \Delta]$ confines n within the scope of the declaration of G. In addition, when T is a channel type, the typing ensures that only data that can flow on G may be sent on n.

The structural type conveys information on the value, i.e., whether it is basic value, in which case the structural type is a base type B, or a channel: as in other systems, a channel type specifies the types τ_i of the payload as well as the capabilities ν associated with the channel.

Finally, the delivery policy defines the groups of the channels on which the value may be exchanged, as well as the type at which the exchange may take place. Thus if $\Delta = \sum_{i \in I} \mathsf{G}_i : T_i \to \Delta_i$, the legal flows of a value of type $\mathsf{G}[T \parallel \Delta]$ are those that reach a channel of group G_i, at type T_i, and then continue to flow as specified by Δ_i, for all $i \in I$. The distinguished group symbol any may be used to create a default entry in Δ. In particular, if $\mathsf{G}_j = \mathsf{any}$ for some $j \in I$, then Δ allows the transmission over channels of any group other than the G_is, provided that the transmissions occurs at the type (and with the policy) associated with the jth entry in Δ. An empty policy, noted by ε, signals that the value should not be (re-)transmitted on any channel: if $\Delta_i = \varepsilon$, we write $\mathsf{G} : T \to \Delta$ simply as $\mathsf{G} : T$. A delivery policy may be recursive, to allow the transmission of values to an unbounded distance. A recursive policy $\mu X.\Delta$ binds the recursion variable X with scope Δ: throughout the paper, we assume that the delivery policies that occur in a resource type are *closed*, i.e., all their recursion variables X are bound by an enclosing binder, and *formally contractive*, in the following sense: $\mu X.\Delta$ is contractive in the variable X if either X does not occur free in Δ, or Δ can be rewritten via unfolding as a choice. The recursion constructor only applies to policies, not to types: while the extension to recursive types would be possible, and harmless, recursive policies are all we need for our present purposes.

2.1. *Operational semantics*

The dynamics of the calculus could be defined in an entirely standard way, in terms of reduction. On the other hand, in order to state and prove the properties of the type system, it is convenient to resort to an instrumented semantics where the flow of each name and value is made explicit along reduction.

To account for that, we use tags to trace the sequence of channels traversed by each name in a process. Thus, for example, we note with $n_{[mpq]}$ the name n that flowed first through the channel m, then through p and finally through q. Here 'mpq' is short for the extended notation $m :: p :: q :: \varepsilon$, where ε denotes the empty sequence and '::' is the right-associative sequence constructor. We let φ range over sequences of names, and use the notation $\varphi \cdot n$ to indicate the sequence resulting from appending the name n to the tail of φ.

The definition of the instrumented semantics is given in Table 1. Formally, it is defined as a binary relation over *dynamic processes* rather than processes. Dynamic processes, ranged over by $A, B, \ldots$ are defined just like processes, but are built over tagged names and values, rather than names and values. The tags may be empty, in which case we write tagged names just as names. In fact, processes themselves are to be understood as special cases of dynamic processes in which all names are untagged.

The notion of free names for dynamic processes is easily adapted to account for the presence of the tags: namely, $fn(n_{[\varphi]}(x_1 : \tau_1, \ldots, x_h : \tau_h).A) = fn(A) \cup \{n\} \cup \{m \mid m \in \varphi\}$, and similarly for the remaining dynamic process constructs. The clauses that define the reduction and structural

Table 1 The flow-sensitive operational semantics

Dynamic Processes: $a, b, \dots$ denote either (non-tagged) variables or tagged names of the form $n_{[\varphi]}$.

$$A, B ::= \mathbf{0} \mid a(x_1{:}\tau_1, \dots, x_n{:}\tau_n).A \mid \overline{a}\langle b_1, \dots, b_n\rangle.A \mid (\nu n : \tau)A \mid (\nu \mathsf{G})A \mid A|B \mid !A$$

Structural congruence

(*monoid*))	$A \mid B \equiv B \mid A, \quad A \mid \mathbf{0} \equiv A, \quad (A \mid B) \mid C \equiv A \mid (B \mid C)$			
(*repl*))	$!A$	$\equiv$	$A \| !A$	
(*name extr*)	$(\nu n : \tau)(A \mid B)$	$\equiv$	$A \mid (\nu n : \tau)B$	if $n \notin fn(A)$
(*group extr*)	$(\nu \mathsf{G})(A \mid B)$	$\equiv$	$A \mid (\nu \mathsf{G})B$	if $\mathsf{G} \notin fg(A)$
(*name/group exch*)	$(\nu \mathsf{G})(\nu n : \tau)A$	$\equiv$	$(\nu n : \tau)(\nu \mathsf{G})A$	if $\mathsf{G} \notin fg(\tau)$
(*name exch*)	$(\nu n_1 : \tau_1)(\nu n_2 : \tau_2)A$	$\equiv$	$(\nu n_2 : \tau_2)(\nu n_1 : \tau_1)A$	$(n_1 \neq n_2)$
(*group exch*)	$(\nu \mathsf{G}_1)(\nu \mathsf{G}_2)A$	$\equiv$	$(\nu \mathsf{G}_2)(\nu \mathsf{G}_1)A$	

Reduction

(*red comm*)	$\overline{n_{[\varphi]}}\langle m_{1\,[\varphi_1]}, \dots, m_{k\,[\varphi_k]}\rangle.A \mid n_{[\psi]}(x_1{:}\tau_1, \dots, x_k{:}\tau_k).B \longrightarrow A \mid B\{{}^{m_{i\,[\varphi_i \cdot n]}}/_{x_i}\}$		
(*red res*)	$A \longrightarrow A'$	$\Longrightarrow$	$(\nu n{:}\tau)A \longrightarrow (\nu n{:}\tau)A'$
(*red group*)	$A \longrightarrow A'$	$\Longrightarrow$	$(\nu \mathsf{G})A \longrightarrow (\nu \mathsf{G})A'$
(*red par*)	$A \longrightarrow A'$	$\Longrightarrow$	$A \mid B \longrightarrow A' \mid B$
(*red struct*)	$A \equiv\longrightarrow\equiv B$	$\Longrightarrow$	$A \longrightarrow B$

congruence relations are largely standard. Indeed, structural congruence is defined just as in πG, though it relies on the definition of free names we just discussed. The core of reduction, in turn, is the synchronization rule (*red comm*), which formalizes semantics of value exchange and additionally updates the tags of the values exchanged so as to trace the flow of each of the arguments through the synchronization channel.

Since values are used non-linearly in processes, the same value may end-up flowing along different paths and hence acquire different tags. For instance, two steps of flow-reduction from the process $\overline{a}\langle m\rangle \mid \overline{b}\langle m\rangle \mid a(x).a(y).P$ yield the dynamic process $P\{{}^{m_{[a]}}/_x, {}^{m_{[b]}}/_y\}$, in which the two instances of m can be distinguished by their tags tracing the paths along which m has flown. Notice, however, that the tags are only functional to analyze the properties of the type system and to establish our safety results. Instead, they have no effect on the reductions available for processes. We make this precise by relating our flow-sensitive semantics of Table 1 with the original reduction semantics of πG. The latter, which we denote with $\mapsto$, is defined on processes (rather than on dynamic processes) exactly as we do here, but uses the standard communication rule

$$\overline{n}\langle m_1, \dots, m_k\rangle.P \mid n(x_1{:}\tau_1, \dots, x_k{:}\tau_k).Q \mapsto P \mid Q\{{}^{m_i}/_{x_i}\},$$

in place of our (*red comm*) rule. Given any dynamic process A, let $|A|$ denote the static process resulting from erasing all the tags from A. Let also $\longrightarrow^*$ and $\mapsto^*$ denote the reflexive and transitive closure of the reduction relations $\longrightarrow$ and $\mapsto$, respectively. Then we have the following, easily proved, result.

Proposition 2.1 (Flow vs Standard reductions). Let A be a closed dynamic process. If $A \longrightarrow^* B$ then $|A| \mapsto^* |B|$. Conversely, if $|A| \mapsto^* Q$ then there exists B such that $A \longrightarrow^* B$ and $|B| \equiv Q$.

3. Principals and discretionary access control policies

Principals are defined on top of the process calculus with groups, by establishing a connection between the group names in the process calculus and the principals, and by construing principals as the unit of abstraction for access control. Among many possible alternatives, we choose the simplest representation of principals, based on a mapping between the group names of the low-level process calculus and principal names.

3.1. *Principals and Systems*

A system is a composition of principals, formed with the following productions:

$$S \ ::= \ \mathsf{G}\{P\} \mid S \parallel S \mid (\nu \mathsf{G})S \mid (\nu n : \tau)S$$

Principals are denoted by terms of the form $\mathsf{G}\{P\}$, where G is a group name acting as the principal identity, and P is the body, a process in the low-level calculus. In a system, principals may run in parallel and share local (group) names.

The principal form $\mathsf{G}\{\cdot\}$ defines a static scope that helps enforce the discipline on the declaration of resources (i.e. names) established by the following notion of well-formedness. Say that a name n is controlled by G if $n : \mathsf{G}[...]$: then the principal $\mathsf{G}\{P\}$ is well-formed if P it only creates new names controlled by G or by any of the local group names that P itself creates with a group declaration. A system is well-formed iff so are all of the component principals: throughout, assume that our systems are well-formed.

The dynamics of systems could be derived from the reduction relation over the underlying processes, by stipulating that $S \longrightarrow S'$ whenever $[S] \longrightarrow R$ and $R = [S']$ (one easily verifies that one such S' exists for all R). However, this would convey principals a computational stand that, instead, we choose to dispense with. Indeed, we interpret principals as purely static entities, that define the access control policies associated with their resources, as we just discussed, but bear no computational meaning. This is consistent with the nature of our resource policies which, once established by a principal, are only intended to discipline the flow of names along the channels of a system, irrespective of the principals on behalf of which the code exchanging those names is run. Consequently, principals may be "compiled away" after the well-formedness check, and their dynamic semantics be defined directly in terms of the computation of their body. Formally, given a system S, let $[S]$ the process obtained from S by transforming each principal $\mathsf{G}\{P\}$ into its body P: then $S \longrightarrow^* R$ just in case $[S] \longrightarrow^* R$.

3.2. *Type Expressions*

In the calculus of principals we presuppose an extended syntax for types to ease the specification of the types themselves and of the associated access control policies. In particular, we write the choices of a policy as a semi-colon separated list of entries, as in $\mathsf{G}[T \parallel \mathsf{G}_1 \to \Delta_1; \dots ; \mathsf{G}_n \to \Delta_n]$. Further, we introduce the high-level type $\mathsf{G}[T]$ corresponding to the low-type $\mathsf{G}[T \parallel \varepsilon]$ whose delivery policy is empty. The calculus of principals also includes the new type expression $\tau \downarrow \pi$, with π a sequence of group names, to note the type occurring in τ at the hop reached via the path π. For instance, if $\tau = \mathsf{G}[T \parallel \mathsf{G}_1 : T_1 \to (\mathsf{G}_2 : T_2 \to \Delta_2 \ ; \ \mathsf{G}_3 : T_3 \to \Delta_3)]$, then $\tau \downarrow \mathsf{G}_1.\mathsf{G}_2 = \mathsf{G}[T_2 \parallel \Delta_2]$.

We illustrate the types at work in the specification of access control policies, starting with a simple example. Let acct be a basic type describing the data (say, name and password) associated with an electronic account that clients may use to sign-up an on-line service. Typically, the account data must be supplied to the e-service at each login; at the same time, however, the clients will want to make sure that the service does not leak their data to third parties. This policy is expressed naturally in our system, by the type $\mathsf{Client}[\mathsf{acct} \parallel \mathsf{Service} : \mathsf{acct}]$, whose policies allows the transmission of the account data only to the service. Generalizing the example, assume now that the client of the on-line service is an organization (say a university department) that has obtained a collective account to be shared by all its members. In that case, the policy for the account data should be more flexible, so as to allow the data to circulate freely within the organization. That policy is expressed by the type $\mathsf{Dept}[\mathsf{acct} \parallel \mu X.(\mathsf{Dept} : \mathsf{acct} \to X;\ \mathsf{Service} : \mathsf{acct})]$: here, the policy allows the free re-transmission of the account data within department channels (mailboxes) while still requiring the service to refrain from passing it on to third parties.

The (re-)transmission of values may also be regulated on the basis of the capabilities that are passed along with the names (channels) transmitted. For instance, a collaborative *wiki* environments on the web typically splits users in two classes, authors and users, with different access rights on the system (read/write and read-only respectively). If we represent the wiki pages as channels, of type $(\mathsf{info})^{\mathsf{rw}}$, then the following type establishes the desired access control policy:

$$\mathsf{Author}[(\mathsf{info})^{\mathsf{rw}} \parallel \mu X.(\mu Y.(\mathsf{User} : (\mathsf{info})^{\mathsf{r}} \to Y)\ ;\ \mathsf{Author} : (\mathsf{info})^{\mathsf{rw}} \to X)]$$

The authors may freely exchange the wiki pages with full access rights; on the other hand, when passed on to users, the wiki pages have read-only access, as expected. The users may themselves advertise the pages to other users, but again with read-only access right.

The presence of recursive policies make our types a strict extension of the types found in πG, which are defined by the following productions $T ::= \mathsf{G}[T_1,\ldots,T_n]$ (for $n = 0$ the production generates the base case $\mathsf{G}[\,]$). These types may be encoded into our resource types as follows:

$$[\![\mathsf{G}[T_1,\ldots,T_n]]\!] = \mathsf{G}[([\![T_1]\!],\ldots,[\![T_n]\!])^{\mathsf{rw}} \parallel \mu X.\mathsf{any} : ([\![T_1]\!],\ldots,[\![T_n]\!])^{\mathsf{rw}} \to X].$$

In other words, all πG types are interpreted as channel types provided with the most liberal delivery policy, one that allows the delivery over all channels: the only constraint is that the receiving channels have access to the group of the value they carry with them, exactly as in (Cardelli et al., 2005).

3.3. *An on-line editorial system*

We further illustrate the calculus of principals, and the expressiveness of our types, with a more extended example, in which we formalize a simple system to handle on-line submissions and reviews for a journal.

To help structure the specification, we introduce a series of macros, specified in Table 2, for type expressions and principal definitions, and then we form the desired system around those macros. The system comprises four main principal identities: Journal, Author, Editor, Reviewer, and three main types of documents exchanged: `MANUSCRIPT`, `REVIEW`, `DECISION`. All such documents come with policies that control their flow during the submission/review process. In particular, the manuscripts are created by their authors and flow on a path that includes the journal first, then the editor and finally the reviewers. The reports written by the reviewers are sent back to the editor, then forwarded first to the journal and then on to the authors. The editor uses the reviews to make an accept/reject decision, which is sent both to the journal, to forward it to the author, and to the reviewers.

Table 2 Macros for the types and and principals of the on-line editorial system

Types

Documents Exchanged

$\mathtt{MANUSCRIPT}$	$=$	$\mathsf{Author}[\mathsf{string} \Vert \mathsf{Journal} : \mathsf{string} \to \mathsf{Editor} : \mathsf{string} \to \mathsf{Reviewer} : \mathsf{string}]$
$\mathtt{REVIEW}$	$=$	$\mathsf{Reviewer}[\mathsf{string} \Vert \mathsf{Editor} : \mathsf{string} \to \mathsf{Journal} : \mathsf{string} \to \mathsf{Author} : \mathsf{string}]$
$\mathtt{DECISION}$	$=$	$\mathsf{Editor}[\mathsf{string} \Vert \mathsf{Journal} : \mathsf{string} \to \mathsf{Author} : \mathsf{string}\ ;\ \mathsf{Reviewer} : \mathsf{string}]$

Main channels employed in the exchanges

$\mathtt{NOTIFYAUTH}$	$=$	$\mathsf{Author}[(\mathtt{DECISION} \downarrow \mathsf{Journal.Author},\ \mathtt{REVIEW} \downarrow \mathsf{Editor.Journal.Author})^{\mathsf{rw}}$ $\Vert\ \mathsf{Journal} : (\mathtt{DECISION} \downarrow \mathsf{Journal.Author},\ \mathtt{REVIEW} \downarrow \mathsf{Editor.Journal.Author})^{\mathsf{w}}]$
$\mathtt{CONFIRM}$	$=$	$\mathsf{Author}[(\mathtt{SUBMIT} \downarrow \mathsf{Author})^{\mathsf{rw}} \Vert \mathsf{Journal} : (\mathtt{SUBMIT} \downarrow \mathsf{Author})^{\mathsf{w}}]$
$\mathtt{NOTIFYJOUR}$	$=$	$\mathsf{Journal}[(\mathtt{DECISION} \downarrow \mathsf{Journal},\ \mathtt{REVIEW} \downarrow \mathsf{Editor.Journal})^{\mathsf{rw}}$ $\Vert\ \mathsf{Editor} : (\mathtt{DECISION} \downarrow \mathsf{Journal},\ \mathtt{REVIEW} \downarrow \mathsf{Editor.Journal})^{\mathsf{w}}]$
$\mathtt{SUBMIT}$	$=$	$\mathsf{Journal}[(\mathtt{MANUSCRIPT} \downarrow \mathsf{Journal},\ \mathtt{NOTIFYAUTH} \downarrow \mathsf{Journal})^{\mathsf{rw}}$ $\Vert\ \mathsf{Author} : (\mathtt{MANUSCRIPT} \downarrow \mathsf{Journal},\ \mathtt{NOTIFYAUTH} \downarrow \mathsf{Journal})^{\mathsf{w}}]$
$\mathtt{REGISTER}$	$=$	$\mathsf{Journal}[(\mathsf{string},\ \mathtt{CONFIRM} \downarrow \mathsf{Journal})^{\mathsf{rw}}]$

Additional channels

$\mathtt{EVALUATE}$	$=$	$\mathsf{Editor}[(\mathtt{REVIEW} \downarrow \mathsf{Editor})^{\mathsf{rw}} \Vert \mathsf{Reviewer} : (\mathtt{REVIEW} \downarrow \mathsf{Editor})^{\mathsf{w}}]$
$\mathtt{HANDLE}$	$=$	$\mathsf{Editor}[(\mathtt{MANUSCRIPT} \downarrow \mathsf{Journal.Editor},\ \mathtt{NOTIFYJOUR} \downarrow \mathsf{Editor})^{\mathsf{rw}}]$
$\mathtt{ASGN}$	$=$	$\mathsf{Reviewer}[(\mathtt{MANUSCRIPT} \downarrow \mathsf{Journal.Editor.Reviewer},\ \mathtt{EVALUATE} \downarrow \mathsf{Reviewer})^{\mathsf{rw}}]$

Principals

$\mathtt{AUTHOR}$	$=$	$\mathsf{Author}\{(\nu confirm{:}\mathtt{CONFIRM})\,\overline{register}\langle info, confirm\rangle.confirm(submit).$ $(\nu na : \mathtt{NOTIFYAUTH})\overline{submit}\langle draft, na\rangle.na(decision, review)\ldots\}$
$\mathtt{JOURNAL}$	$=$	$\mathsf{Journal}\{\ !register(info, confirm).(\nu submit : \mathtt{SUBMIT})\,\overline{confirm}\langle submit\rangle.$ $submit(draft, na).(\nu nj : \mathtt{NOTIFYJOUR})\,\overline{handle}\langle draft, nj\rangle.nj(dec, rep).\overline{na}\langle dec, rep\rangle\}$
$\mathtt{EDITOR}$	$=$	$\mathsf{Editor}\{\ !handle(m,n).(\nu e : \mathtt{EVALUATE})\overline{asgna}\langle m, e\rangle \mid \overline{asgnb}\langle m, e\rangle \mid$ $e(reva).e(revb). <\text{make a decision} d : \mathtt{DECISION} > .\overline{n}\langle d, reva :: revb\rangle\}$
$\mathtt{REVIEWERA}$	$=$	$\mathsf{Reviewer}\{\ !asgna(m,e). <\text{read and produce} review : \mathtt{REVIEW} > .\overline{e}\langle review\rangle\ \}$
$\mathtt{REVIEWERB}$	$=$	$\mathsf{Reviewer}\{\ !asgnb(m,e). <\text{read and produce} review : \mathtt{REVIEW} > .\overline{e}\langle review\rangle\ \}$

An instance of the entire system may be obtained as the parallel composition of the principals obtained by (recursively) expanding the principal and type macros in the term below:

$$\mathtt{AUTHOR} \Vert \mathtt{JOURNAL} \Vert \mathtt{EDITOR} \Vert \mathtt{REVIEWERA} \Vert \mathtt{REVIEWERB}$$

The author principal first registers to the journal and then submits his/her papers; the journal principal represents the on-line system interfacing with authors and editors; the editor principal interacts with the journal; finally, the reviewers read the manuscripts and report back to the editor. To ease the notation, we omit the type annotations on the input variables, as they are directly inferred by the type of the input channels.

The channel types describe (and constrain) the policies governing the dataflow in the submission and review protocols. The authors access the system via the *register* channel; the registration is finalized when the system reports back with an ack on the *confirm* channel: notice that each author retains full access on this channel that is instead delivered with write-only access to the journal. Registered authors have confidentiality guarantees that their manuscripts will not be distributed to readers other than the editorial system staff (editors and reviewers), and that notifications about their papers will not reach any principals other than themselves outside the editorial system. The first guarantee derives from the structure of the $\mathtt{MANUSCRIPT}$ type. The second is a consequence of (i) the policy defined by the type $\mathtt{DECISION}$, which requires that notifications be only sent to the $\mathsf{Journal}$ and $\mathsf{Reviewers}$, and (ii) of the structure of the journal principal that forwards the decision only to the legitimate author. Moreover, the type $\mathtt{REVIEW}$ guarantees that reviewers may not consult each other by exchanging reviews.

Clearly, other specifications are possible for the system. For instance, the journal may wish to shield the internal processing of a submission from external observations. In that case, the specification should be structured as follows

$$\mathtt{AUTHOR} \parallel (\nu\mathtt{Editor})(\nu\mathtt{Reviewer})(\mathtt{JOURNAL} \parallel \mathtt{EDITOR} \parallel \mathtt{REVIEWERA} \parallel \mathtt{REVIEWERB})$$

The system now splits the component principals into two separate scopes. Inside the scope of the group declaration, all principals are known, while outside the scope, Author has only visibility of $\mathsf{Journal}$[†]. Of course, this requires some changes in the way that the flow policies are designed inside the types and the principals of the system. For instance, authors should be prepared to accept weaker guarantees for their manuscripts, as expressed by the following type:

$$\mathtt{MANUSCRIPT} = \mathsf{Author}[\mathsf{string} \parallel \mathsf{Journal} : \mathsf{string} \to (\mu X.\mathsf{any} : \mathsf{string} \to X)]$$

This policy is still compatible with the intended flow of the manuscript, but it only mentions the $\mathsf{Journal}$ principal, leaving it in full control on how the manuscripts are circulated.

Having split the system into two scopes, the code of principals and the policies associated with their resources must also be restructured to break the flow of such resources across the two scopes when needed. Specifically, the journal may not simply forward decisions and reviews as it receives them from the editor and reviewers; instead, it will have to notify the authors directly, producing its own decision based on the data it receives from the principals inside the restricted scope. The types establishing the policies for reviews and decisions must be redefined accordingly, to stop the flow of their data at the journal principal:

$$\begin{array}{lcl} \mathtt{REVIEW} & = & \mathsf{Reviewer}[\mathsf{string} \parallel \mathsf{Editor} : \mathsf{string} \to \mathsf{Journal} : \mathsf{string}] \\ \mathtt{DECISION} & = & \mathsf{Editor}[\mathsf{string} \parallel \mathsf{Journal} : \mathsf{string}\ ;\ \mathsf{Reviewer} : \mathsf{string}] \end{array}$$

Then, new types must be introduced to establish the flow of the journal decision and of the reports that the journal sends back to the author

[†] We are implicitly assuming a standard macro expansion mechanism, which may cause variable capture, so that occurrences of the group names Editor and $\mathsf{Reviewer}$ get captured by the group restriction in the expansion of of the type and principal macros.

`JDECISION`= $\mathsf{Journal}[\mathsf{string} \parallel \mathsf{Author} : \mathsf{string}]$ `JREVIEW`= $\mathsf{Journal}[\mathsf{string} \parallel \mathsf{Author} : \mathsf{string}]$

4. The Type System

The type system is based on the syntax of types from the calculus of processes. As we shall see, the typing of principals is derived easily from the typing rules for processes by rewriting the type expressions $\tau \downarrow \pi$ used in the calculus of principals into their (the types') low-level counterpart. In that direction, we formalize the interpretation of the type expressions $\tau \downarrow \pi$. First define the notation $\Delta(\mathsf{G}) = T \to \Delta'$ to identify the entry in Δ relative to the group G (if any):

$$\begin{array}{rcl} \varepsilon(\mathsf{G}) & = & X(\mathsf{G}) = \mathsf{undefined} \\ (\Sigma_i\, \mathsf{G}_i : T_i \to \Delta_i)(\mathsf{G}) & = & \mathsf{if}\ (\exists j.\ \mathsf{G} = \mathsf{G}_j)\ \mathsf{then}\ T_j \to \Delta_j \\ & & \mathsf{else\ if}\ (\exists k.\ \mathsf{G}_k = \mathsf{any})\ \mathsf{then}\ T_k \to \Delta_k\ \mathsf{else\ undefined} \\ (\mu X.\Delta)(\mathsf{G}) & = & \Delta\{\mu X.\Delta/X\}(\mathsf{G}) \end{array}$$

Then we view a policy as a partial function, and we define $Dom(\Delta) \triangleq \{\mathsf{G} : \Delta(\mathsf{G})\ \text{is defined}\}$. Finally we define the evaluation of the type expression $\tau \downarrow \pi$ by interpreting $\downarrow$ as the partial operator given below:

$$\begin{array}{rcl} \tau \downarrow \varepsilon & \triangleq & \tau \\ \mathsf{G}[T \parallel \Delta] \downarrow (\overline{\mathsf{G}}.\pi) & \triangleq & \mathsf{if}\ \Delta(\overline{\mathsf{G}}) = \overline{T} \to \overline{\Delta}\ \mathsf{then}\ \mathsf{G}[\overline{T} \parallel \overline{\Delta}] \downarrow \pi\ \mathsf{else\ undefined} \end{array}$$

4.1. *Type and environment formation*

We assume type equality up to (*i*) renaming of the bound variables in a policy, (*ii*) permutation of entries inside policies (e.g. $\mathsf{G}[T \parallel \mathsf{G}_1 : T_1 \to \Delta_1;\ \mathsf{G}_2 : T_2 \to \Delta_2] = \mathsf{G}[T \parallel \mathsf{G}_2 : T_2 \to \Delta_2;\ \mathsf{G}_1 : T_1 \to \Delta_1]$), and (*iii*) unfolding of recursive policies (i.e., $\mathsf{G}[T \parallel \mu X.\Delta] = \mathsf{G}[T \parallel \Delta\{\mu X.\Delta\}/X]$). To ease the notation, we often write $\Delta\{X\}$ for a policy where X occurs free, and $\Delta\{\Delta'\}$ for the result of substituting X with Δ' in Δ.

Type formation is defined in Table 3. The rules for structural types are standard. As for resource types, the rules enforce two constraints, namely: a type $\mathsf{G}[T \parallel \Delta]$ should only mention group names known to the type environment Γ, and furthermore all the structural types occurring in Δ should be super-types of type T. The first constraint helps guarantee a secrecy property à la πG; the second, in turn, is crucial for soundness and is enforced by the type formation rules with the help of the auxiliary judgments $\Gamma \vdash_T \Delta$, that check the delivery constraints in Δ against type T.

The rules for environments are essentially as in (Cardelli et al., 2005). A well-formed environment contains group names, typing assumptions for variables and names, and policy variables. All names in an environment are untagged, and all group names in the type of a variable/name must be declared in the environment.

4.2. *Subtyping*

The subtyping relation is defined by the rules in Table 4. They derive judgments of the form $\Sigma \vdash \tau \leq \tau'$, where Σ is a set of ordering assumptions $\Delta \leq \Delta'$ on policies. When Σ is empty, we write

Table 3 Good types and environments

Environments

(EMPTY)
$$\frac{}{\varnothing \vdash \diamond}$$

(ENV n)
$$\frac{\Gamma \vdash \diamond \quad \Gamma \vdash \tau \quad n \notin Dom(\Gamma)}{\Gamma, n : \tau \vdash \diamond}$$

(ENV x)
$$\frac{\Gamma \vdash \diamond \quad fg(\tau) \subseteq Dom(\Gamma) \quad x \notin Dom(\Gamma)}{\Gamma, x : \tau \vdash \diamond}$$

(ENV G)
$$\frac{\Gamma \vdash \diamond \quad \mathsf{G} \notin Dom(\Gamma)}{\Gamma, \mathsf{G} \vdash \diamond}$$

(ENV X)
$$\frac{\Gamma \vdash \diamond \quad X \notin Dom(\Gamma)}{\Gamma, X \vdash \diamond}$$

Structural and Resource Types

(BASE)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathsf{B}}$$

(CHANNEL)
$$\frac{\Gamma \vdash \tau_i \quad i = 1,..,n}{\Gamma \vdash (\tau_1, \dots, \tau_n)^{\mathsf{v}}}$$

(τ-OWNER)
$$\frac{\Gamma \vdash T \quad \Gamma \vdash_T \Delta \quad \mathsf{G} \in Dom(\Gamma)}{\Gamma \vdash \mathsf{G}[T \parallel \Delta]}$$

Delivery Policies

(Δ-VAR)
$$\frac{X \in \Gamma \quad \Gamma \vdash \diamond}{\Gamma \vdash_T X}$$

(Δ-REC)
$$\frac{\Gamma, X \vdash_T \Delta\{X\}}{\Gamma \vdash_T \mu X.\Delta\{X\}}$$

(Δ-EMPTY)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash_T \varepsilon}$$

(Δ-STEP)
$$\frac{\mathsf{G}_i \in Dom(\Gamma) \quad \vdash T \leq T_i \quad \Gamma \vdash_T \Delta_i \quad \Gamma \vdash T_i \quad \forall i \in I}{\Gamma \vdash_T \sum_{i \in I} G_i : T_i \to \Delta_i}$$

$\tau' \leq \tau$ and $\vdash \tau' \leq \tau$ interchangeably. We overload the $\leq$ symbol to denote the subtype relation, the order on policies as well as the partial order on capabilities, defined as usual to be the reflexive closure of the relation defined by $\mathsf{rw} \leq \mathsf{r}$, $\mathsf{rw} \leq \mathsf{w}$. Subtyping over resource types is defined by the (τ-TYPE/POLICY) rule, which makes resource-subtyping covariant in the component structural types and in the ordering relationship over policies.

The ordering on policies constitutes the core of the subtype system. It is reminiscent of the subtype relation on record types, and indeed, the rationale is the same: in any context where we expect a value that has a certain set of enabled flows, it is safe to receive a value which has additional legal flows. Rule (SUB-Δ-1) captures this idea for policies without the default entry any. For policies containing the default entry the details are subtler. Rule (SUB-Δ-2) states that a higher policy may include an entry for a group G_i not occurring in a lower policy, but only when the lower policy contains the default entry with a compatible policy. On the other hand, if a policy contains a default entry, then all lower policies must contain a default entry as well. Rule (SUB-Δ-3) handles this case: it generalizes rule (SUB-Δ-1) and further requires the policies for the entries G_i in the lower policy that do not occur in the higher policy to match the default entry of the higher policy. Thus, for instance $(\mathsf{any} : \tau^{\mathsf{rw}}) \leq (\mathsf{G}_1 : \tau^{\mathsf{r}} + \mathsf{G}_2 : \tau^{\mathsf{w}})$ by rule (SUB-Δ-2), as both the entries in the higher policy are safely supported by the default entry in the lower policy. On the other hand $(\mathsf{G}_1 : \tau^{\mathsf{r}} + \mathsf{any} : \tau^{\mathsf{rw}}) \not\leq \mathsf{any} : \tau^{\mathsf{rw}}$ as the lower policy includes an entry G_1 with

Table 4 The subtype relation

Structural subtypes

(*T*-Reflex)
$$\frac{}{\Sigma \vdash T \;\leq\; T}$$

(*T*-Read)
$$\frac{\mathsf{v} \leq \mathsf{r} \quad \Sigma \vdash \tau_1 \leq \tau_1' \quad \cdots \quad \Sigma \vdash \tau_n \leq \tau_n'}{\Sigma \vdash (\tau_1,\ldots,\tau_n)^{\mathsf{v}} \;\leq\; (\tau_1',\ldots,\tau_n')^{\mathsf{r}}}$$

(*T*-Write)
$$\frac{\mathsf{v} \leq \mathsf{w} \quad \Sigma \vdash \tau_1' \leq \tau_1 \quad \cdots \quad \Sigma \vdash \tau_n' \leq \tau_n}{\Sigma \vdash (\tau_1,\ldots,\tau_n)^{\mathsf{v}} \;\leq\; (\tau_1',\ldots,\tau_n')^{\mathsf{w}}}$$

Resource Subtypes

(τ-Reflex)
$$\frac{}{\Sigma \vdash \tau \leq \tau}$$

(τ-Type/Policy)
$$\frac{\Sigma \vdash T \leq T' \quad \Sigma \vdash \Delta \leq \Delta'}{\Sigma \vdash \mathsf{G}[T \parallel \Delta] \leq \mathsf{G}[T' \parallel \Delta']}$$

Recursive Sub-Policies

(Δ-Reflex)
$$\frac{}{\Sigma \vdash \Delta \leq \Delta}$$

(Δ-Assume)
$$\frac{}{\Sigma,\ \Delta_1 \leq \Delta_2,\ \Sigma' \vdash \Delta_1 \leq \Delta_2}$$

(Δ-Left Unfold)
$$\frac{\Sigma, \mu X.\Delta_1\{X\} \leq \Delta_2 \vdash \Delta_1\{\mu X.\Delta_1\{X\}\} \leq \Delta_2}{\Sigma \vdash \mu X.\Delta_1\{X\} \leq \Delta_2}$$

(Δ-Right Unfold)
$$\frac{\Sigma, \Delta_1 \leq \mu X.\Delta_2\{X\} \vdash \Delta_1 \leq \Delta_2\{\mu X.\Delta_2\{X\}\}}{\Sigma \vdash \Delta_1 \leq \mu X.\Delta_2\{X\}}$$

(Sub-Δ-1)
$$\frac{\Sigma \vdash T_i \leq T_i' \quad \Sigma \vdash \Delta_i \leq \Delta_i' \quad \mathsf{G}_i \neq \mathsf{any} \quad i \in I \cup J}{\Sigma \vdash \sum_{i \in I \cup J} \mathsf{G}_i : T_i \to \Delta_i \;\leq\; \sum_{i \in I} \mathsf{G}_i : T_i' \to \Delta_i'}$$

(Sub-Δ-2)
$$\frac{\begin{array}{l} \mathit{Dom}(\Delta') \subseteq \mathit{Dom}(\Delta) \quad \Sigma \vdash \Delta \leq \Delta' \quad \mathsf{any} \notin \mathit{Dom}(\Delta') \cup \{\mathsf{G}_i\}_{i \in I} \\ \mathit{Dom}(\Delta) \cap \{\mathsf{G}_i\}_{i \in I} = \emptyset \quad (\Sigma \vdash T_d \leq T_i \quad \Sigma \vdash \Delta_d \leq \Delta_i \quad \forall i \in I) \end{array}}{\Sigma \vdash (\Delta + \mathsf{any} : T_d \to \Delta_d) \;\leq\; (\Delta' + \sum_{i \in I} \mathsf{G}_i : T_i \to \Delta_i)}$$

(Sub-Δ-3)
$$\frac{\begin{array}{l} \mathit{Dom}(\Delta) \subseteq \mathit{Dom}(\Delta') \quad \Sigma \vdash T_d \leq T_d' \quad \Sigma \vdash \Delta_d \leq \Delta_d' \quad \Sigma \vdash \Delta + \mathsf{any} : T_d \to \Delta_d \leq \Delta' \\ \mathit{Dom}(\Delta') \cap \{\mathsf{G}_i\}_{i \in I} = \emptyset \quad (\Sigma \vdash T_i \leq T_d' \quad \Sigma \vdash \Delta_i \leq \Delta_d' \quad \forall i \in I) \end{array}}{\Sigma \vdash (\Delta + \mathsf{any} : T_d \to \Delta_d + \sum_{i \in I} \mathsf{G}_i : T_i \to \Delta_i) \;\leq\; (\Delta' + \mathsf{any} : T_d' \to \Delta_d')}$$

a higher type than that expected by the higher policy (this breaks the premises of (Sub-Δ-3), the only applicable rule in this case).

The remaining rules in table 4 are largely standard. The rules for recursive policies are presented in a co-inductive style and, consequently (Gapeyev et al., 2002), the presentation does not include a general transitivity rule for types and policies (but see Section 6.3).

Collectively, the ordering on types and policies yields a rather flexible system: notice, to this regard, that the subtype relation enables the delivery of values at types that may vary non-monotonically along their delivery chains. This, in turn makes it possible to encode interesting access policies such as those found in the so-called *Liberal DAC* models (Samarati and di Vimercati, 2001; Sandhu and Munawer, 1998). These are based on a decentralized authorization structure by which the owner of an object delegates other users the right to specify authorizations, possibly with the ability of further delegating it. In particular, in the so called *originator controlled* policies (McCollum et al., 1990), the owner retains control over its delegates and the way these grant permissions. An example of such policies is given by the type below, where Owner specifies how a given resource should be distributed to two principals, Alice and Carl:

$$\mathsf{Owner}[(T)^{\mathsf{rw}} \parallel \mathsf{Alice} : (T)^{\mathsf{r}} \;\rightarrow\; \mathsf{Carl} : (T)^{\mathsf{rw}} \;\rightarrow\; \mathsf{Alice} : (T)^{\mathsf{w}}]$$

A channel with this type must first be delivered, in read mode, to Alice, who is delegated to re-transmit it to Carl with the additional write capability; only then is Alice allowed to receive the write capability from Carl. This delivery policy is imposed by the owner to ensure that Alice will not write on the new channel until it has been received also by Carl.

A similar example can be recovered from the literature on cryptographic protocols. Consider the case where two parties, Alice and Bob, wish to establish a private exchange. To accomplish that, Alice creates a fresh name, say c_{AB}, sends it to a trusted Server and delegates it to forward the name to Bob so that the exchange may take place. Here, the Server should only act as a forwarder, and not interfere with the exchanges between Alice and Bob. This can be achieved by the typing

$$c_{AB} : \mathsf{Alice}[(\mathsf{data})^{\mathsf{rw}} \parallel \mathsf{Server} : (\mathsf{data})^{\mathsf{r}} \rightarrow \mathsf{Bob} : (\mathsf{data})^{\mathsf{rw}}]$$

in which the Server receives the channel in read-only mode to protect the integrity of the message forwarded to Bob.

One may wonder how policies such as the ones we just described may be implemented in practice, as they require principals to forward capabilities that they do not possess. Indeed, in a trusted environment, we may rely on the typing system to protect against any misuse of the capabilities, by typing all principals. In an adversarial setting, instead, a capability may be realized as a secret protected by encryption so the intended recipient can access it, while the forwarders can not (cf. (Bugliesi and Giunti, 2007)).

4.3. *Typing of processes*

The typing rules for processes, in Table 5, complete the presentation of the type system. As usual the system is meant to provide static guarantees of soundness, hence the rules are intended for (static) processes. On the other hand, our flow-sensitive semantics yields dynamic processes, and then, to state and prove subject reduction we must devise typing rules for processes as well as dynamic processes. Since the latter include the former (via the mapping that erases the flow annotations) we give just one set of rules, for dynamic processes.

We have three typing rules for names: (PROJECT) and (SUBSUMPTION) and (DELIVERY). Collectively, they allow each name $n : \tau$ (indeed, each instance n_φ of n) to be typed at (any super-type of) the type τ known to the environment, as well as at any of the types mentioned in the

Table 5 Typing of dynamic processes.

(PROJECT) $$\frac{\Gamma, a:\tau \vdash \diamond}{\Gamma, a:\tau \vdash a:\tau}$$

(SUBSUMPTION) $$\frac{\Gamma \vdash a:\tau' \quad \vdash \tau' \leq \tau}{\Gamma \vdash a:\tau}$$

(DELIVERY) $$\frac{\Gamma \vdash n_{[\varphi]} : \mathsf{G}[T \parallel \Delta] \quad \Gamma \vdash m : \mathsf{G}_m[T_m \parallel \Delta_m] \quad \Delta(\mathsf{G}_m) = T_1 \to \Delta_1}{\Gamma \vdash n_{[\varphi m]} : \mathsf{G}[T_1 \parallel \Delta_1]}$$

(INPUT) $$\frac{\Gamma \vdash a : \mathsf{G}[(\tau_1,\ldots,\tau_k)^{\mathsf{r}} \parallel \Delta] \quad \Gamma, x_1:\tau_1,\ldots,x_k:\tau_k \vdash A \quad 0 \leq k}{\Gamma \vdash a(x_1:\tau_1,\ldots,x_k:\tau_k).A}$$

(OUTPUT) $$\frac{\Gamma \vdash a : \mathsf{G}[(\tau_1,\ldots,\tau_k)^{\mathsf{w}} \parallel \Delta] \quad \Gamma \vdash A \quad (\Gamma \vdash b_i : \widehat{\tau}_i \quad \widehat{\tau}_i \downarrow \mathsf{G} = \tau_i \quad \forall i \in [0..k])}{\Gamma \vdash \overline{a}\langle b_1,\ldots,b_k\rangle.A}$$

(NEW G) $$\frac{\Gamma, \mathsf{G} \vdash A}{\Gamma \vdash (\nu \mathsf{G})A}$$

(NEW n) $$\frac{\Gamma, n:\tau \vdash A}{\Gamma \vdash (\nu n:\tau)A}$$

(PAR) $$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \mid B}$$

(DEAD) $$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \mathbf{0}}$$

(REPL) $$\frac{\Gamma \vdash A}{\Gamma \vdash !A}$$

delivery policy associated with τ. As we remarked earlier on, all names occur untagged in the environment of the typing judgements. Thus, in the (PROJECT) rule, when a is a variable, all occurrences of a denote the same symbol. Instead, when projecting a name, the rule should be read as follows:

$$\frac{\Gamma, n:\tau \vdash \diamond}{\Gamma, n:\tau \vdash n_\varepsilon : \tau}$$

A name with a composite tag may then be typed by repeated applications of the (DELIVERY) rule. This is the way the type system allows a name to be typed at all its delivery types, a property which is crucial in the proof of subject reduction. To see why, consider rule (OUTPUT), the core of the delivery discipline. Let m be emitted on a channel, say $n : \mathsf{G}[(\tau)^{\mathsf{w}}]$, and assume that m is known to the environment at the type $\widehat{\tau} = \mathsf{F}[T \parallel \Delta]$. The condition $\widehat{\tau} \downarrow \mathsf{G} = \tau$ in rule (OUTPUT) verifies that G is indeed one of the next 'hops' in Δ, and that the type at which m should be delivered to G is τ, the type of values carried by n. Given that the types of the names known to the environment must be well-formed, the type formation rules ensure that the original structural type of m is a subtype of the structural component of $\widehat{\tau}$, thus also a subtype of the type at which n expects to receive its values: this guarantees that m may safely be received at n. Further re-transmissions of m will undergo the same checks by the (OUTPUT) rule, but now with the types advertised at the subsequent hops in Δ: in order to prove subject reduction we therefore need to be able to type m at all such types.

We show the effect of value propagation in the typing system with a typed version of the print spooler example discussed in the introduction (the same effect can be verified on the more complex example of the on-line editorial system of Section 3). Let

$$\mathsf{JOB} = \mathsf{Client}[\mathsf{data} \parallel \mathsf{Spooler} : \mathsf{data} \to \mathsf{Printer} : \mathsf{data} \to \mathsf{Client} : \mathsf{data}]$$

and consider the following composite process:

$$\begin{array}{l}(\nu\, j{:}\mathsf{JOB})\overline{s}\langle j\rangle.ack(x{:}\mathsf{JOB} \downarrow \mathsf{Client})\ldots.\\ \qquad \mid\ !s(x{:}\mathsf{JOB} \downarrow \mathsf{Spooler}).\overline{print}\langle x\rangle\\ \qquad\qquad \mid\ !print(x : \mathsf{JOB} \downarrow \mathsf{Printer}).(\ldots \mid \overline{ack}\langle x\rangle)\end{array}$$

Here, the client creates a new job j, sends it to the printer and waits for notification. If we assume the typings $s : \mathsf{Spooler}[(\mathsf{JOB} \downarrow \mathsf{Spooler})^{\mathsf{rw}}]$, $print : \mathsf{Printer}[(\mathsf{JOB} \downarrow \mathsf{Printer})^{\mathsf{rw}}]$, and $ack : \mathsf{Client}[(\mathsf{JOB} \downarrow \mathsf{Client})^{\mathsf{rw}}]$, it is a routine check to verify that all types involved are well-formed and that the process type-checks. Notice in particular, how the type of j must change in order to match the types that annotate the input variables of the channels along which j is delivered.

If names were used linearly in the calculus, the effect of rule (DELIVERY) could be achieved more directly by changing the typing of m in Γ, in ways similar to what is done in, e.g. *session types* (Honda et al., 1998). In our system, instead, multiple occurrences of the same name may co-exist in a process, and have different types: we employ the tags to keep track of all such types.

We remark, however, that rule (DELIVERY) is not needed for the typing of static processes, as the rule only applies to names with a non-empty tag and these arise only dynamically as a result of reduction. In fact, the typing system for static processes can be recovered from the system we just presented by dropping rule (DELIVERY) and interpreting all names and values as untagged (hence all processes as static processes). We note $\vdash_S$ the typing relation for static processes, and write $\Gamma \vdash_S P$ to note a derivable judgement in this system. For the reason we just discussed we have:

Lemma 4.1. $\Gamma \vdash_S P$ if and only if $\Gamma \vdash P$ for all static processes P.

4.4. *Typing of principals and systems*

As anticipated, the typing rules for the calculus of principals derive directly from the rules for static processes we have illustrated. They are reported below.

$$\text{(PRINCIPAL)}\ \frac{\mathsf{G}\{P\}\ \textit{well-formed} \quad \Gamma \vdash_S \ulcorner P \urcorner}{\Gamma \vdash \mathsf{G}\{P\}} \qquad \text{(COMP)}\ \frac{\Gamma \vdash S_2 \quad \Gamma \vdash S_1}{\Gamma \vdash S_1 \parallel S_2} \qquad \text{(NEWG)}\ \frac{\Gamma, \mathsf{G} \vdash S}{\Gamma \vdash (\nu \mathsf{G})S} \qquad \text{(NEW)}\ \frac{\Gamma, n : \ulcorner \tau \urcorner \vdash S}{\Gamma \vdash (\nu n : \tau)S}$$

We note with $\ulcorner \tau \urcorner$ the resource type obtained by evaluating each type expression $\tau' \downarrow \pi$ occurring within τ as defined earlier in this section. We also note with $\ulcorner P \urcorner$ the process obtained by replacing each type τ with $\ulcorner \tau \urcorner$ in P. Notice in particular that, $\downarrow$ being a partial operator, so is the transformation $\ulcorner \cdot \urcorner$. We tacitly assyme that all type expressions are well-defined in a well-typed system.

A principal is well-typed if so is its body, and in addition, the body respects the well-formedness constraint imposing that P only creates new names controlled by G or by any of the local group names that P itself creates with a group declaration. In a composite system, each principal has visibility (via Γ) of all the free group names occurring in all the other principals of the composition. The restriction operator is typed just as in the underlying process calculus.

5. Type safety

We continue the presentation of the type system by elucidating its main properties. To state such properties, we introduce one additional (partial) operator on type environments. Given a type environment Γ and a sequence of names φ we let $\Gamma[\varphi]$ denote the corresponding sequence of groups obtained from the types that Γ associates with the names in φ. Inductively:

$$\begin{array}{rcl}\Gamma[\varepsilon] & \triangleq & \varepsilon \\ \Gamma[n :: \varphi] & \triangleq & \text{if } (\Gamma(n) = \mathsf{G}[T \parallel \Delta] \text{ and } \Gamma[\varphi] \neq \bot) \text{ then } \mathsf{G} :: \Gamma[\varphi] \text{ else } \bot\end{array}$$

The intuition here is that φ is the sequence of channels traversed by a name along its flow, and $\Gamma[\varphi]$ is the corresponding sequence of groups. Thus, given a tagged name n_φ, $\Gamma[\varphi]$ expresses the flow of n in terms of the groups of the channels n flows through: that information can be used to assess the safety of these flows against the delivery policy that Γ associates with n.

In the rest of this section we state and prove the safety properties of the type system based on these intuitions. Theorem 5.1 shows that in all well-typed dynamic processes, any access to a channel is permitted by the type of the channel, and the arguments are passed over the channel at subtypes of the expected types. Theorem 5.2 shows that in all well-typed dynamic processes names flow according to the delivery policies expressed by their types. Finally, Theorem 5.3 shows that such properties are preserved by reduction. Collectively, these result provide static guarantees that in any computation spawn from a well-typed, system (hence static process), all resources are accessed and delivered according to the policies defined by their types (cf. Corollary 5.1).

Another approach that often adopted in the statement and proof of type safety (see, for instance, (Hennessy and Riely, 2002a)) relies on the definition of an explicit notion of error and on a proof that well-typed terms do not have error transitions. We chose differently, and stated our safety properties directly in terms of types. The two approaches are equivalent, we find ours technically more elegant and concise.

5.1. *Preliminary lemmas*

Lemma 5.1. Assume $\Gamma \vdash n_{[\varphi]} : \mathsf{G}_1[T_1 \parallel \Delta_1]$. Then $\Gamma(n) = \mathsf{G}[T \parallel \Delta]$ with $\mathsf{G}_1 = \mathsf{G}$, $T \leq T_1$, and $\Gamma \vdash_T \Delta_1$.

Proof. By induction on φ. The proof requires the following result which follows by induction of the derivation of $\Delta \leq \Delta'$:

$$\text{if } \Gamma \vdash_T \Delta \text{ and } \Delta \leq \Delta', \text{ then } \Gamma \vdash_T \Delta' \qquad (1)$$

We proceed with our inductive proof on φ.

case $\varphi = \varepsilon$**.** The hypothesis is $\Gamma \vdash n : \mathsf{G}_1[T_1 \parallel \Delta_1]$, that must have come by (SUBSUMPTION) from $\Gamma(n) = \mathsf{G}[T \parallel \Delta] \leq \mathsf{G}_1[T_1 \parallel \Delta_1]$, hence $\mathsf{G}_1 = \mathsf{G}$ and $T \leq T_1$, thus $\Gamma \vdash_T \Delta_1$ by (1).

case $\varphi = \bar{\varphi} \cdot m$**.** The hypothesis must have come by a number of (SUBSUMPTION) from $\Gamma \vdash n_{[\varphi]} : \mathsf{G}_1[T_1' \parallel \Delta_1']$ with $\mathsf{G}_1[T_1' \parallel \Delta_1'] \leq \mathsf{G}_1[T_1 \parallel \Delta_1]$, and (DELIVERY) from $\Gamma \vdash n_{[\bar{\varphi}]} : \mathsf{G}_1[T' \parallel \Delta']$, $\Gamma \vdash m : \mathsf{F}[T_F \parallel \Delta_F]$, and $\mathsf{F} : T_1' \to \Delta_1' \in \Delta'$ or $\mathsf{any} : T_1' \to \Delta_1' \in \Delta'$. From $\Gamma \vdash n_{[\bar{\varphi}]} : \mathsf{G}_1[T' \parallel \Delta']$, by induction we have $\Gamma(n) = \mathsf{G}[T \parallel \Delta]$ with $\mathsf{G}_1 = \mathsf{G}$, $T \leq T'$ and $\Gamma \vdash_T \Delta'$. The last judgment, together with $\mathsf{F} : T_1' \to \Delta_1' \in \Delta'$ (or $\mathsf{any} : T_1' \to \Delta_1' \in \Delta'$), gives $T \leq T_1'$ and $\Gamma \vdash_T \Delta_1'$. Now, from $\mathsf{G}_1[T_1' \parallel \Delta_1'] \leq \mathsf{G}_1[T_1 \parallel \Delta_1]$ we have $T_1' \leq T_1$ and $\Delta_1' \leq \Delta_1$, hence $T \leq T_1'$, and we conclude $\Gamma \vdash_T \Delta_1$ by (1).

Notice that the statement of the lemma does not explicitly relate Δ and Δ_1. However, we have that Δ is the initial delivery policy for the name n, whereas Δ_1 is the policy for $n_{[\varphi]}$, that is the residual policy for n after the first delivery steps described by φ. □

As a corollary of the previous lemma we have:

Lemma 5.2. Let be $\Gamma \vdash n_{[\varphi]} : \mathsf{G}_1[(\tau_1, \ldots, \tau_k)^{\nu_1} \parallel \Delta_1]$ and $\Gamma \vdash n_{[\psi]} : \mathsf{G}_2[(\rho_1, \ldots, \rho_h)^{\nu_2} \parallel \Delta_2]$. Then $h = k$ and one has $\Gamma \vdash n : \mathsf{G}[T \parallel \Delta]$ with $\mathsf{G} = \mathsf{G}_1 = \mathsf{G}_2$, $T \leq (\tau_1, \ldots, \tau_k)^{\nu_1}$ and $T \leq (\rho_1, \ldots, \rho_h)^{\nu_2}$.

We need two further (standard) lemmas in the proof of subject reduction.

Lemma 5.3.

— *Subject Congruence*: If $\Gamma \vdash A$ and $A \equiv B$, then $\Gamma \vdash B$.

— *Substitution*: If $\Gamma, x : \rho \vdash A$ and $\Gamma \vdash n_{[\varphi]} : \rho$, then $\Gamma \vdash A\{^{n_{[\varphi]}}/_x\}$.

Proof. Subject Congruence has a standard proof: we prove the two statements *(i)* $\Gamma \vdash A$ *and* $A \equiv B$ *imply* $\Gamma \vdash B$, and *(ii)* $\Gamma \vdash A$ *and* $B \equiv Q$ *imply* $\Gamma \vdash B$, by simultaneous induction on the derivations of $A \equiv B$ and $B \equiv A$.

As to Substitution, we must prove the following two statements:

1 If $\Gamma, x : \rho \vdash a : \tau$ and $\Gamma \vdash n_{[\varphi]} : \rho$, then $\Gamma \vdash a\{^{n_{[\varphi]}}/_x\} : \tau$.

2 If $\Gamma, x : \rho \vdash A$ and $\Gamma \vdash n_{[\varphi]} : \rho$, then $\Gamma \vdash A\{^{n_{[\varphi]}}/_x\}$.

Point 1 follows by induction on the derivation of $\Gamma, x : \rho \vdash a : \tau$ and a case analysis on the last rule applied. The proof uses the following standard strengthening property:

$$\Gamma, x : \rho \vdash a : \tau \text{ and } x \neq a \text{ then } \Gamma \vdash a : \tau. \tag{2}$$

(PROJECT) If $a = x$ then $\rho = \tau$. The thesis is $\Gamma \vdash n_{[\varphi]} : \rho$ which is true from the hypothesis. Otherwise, if $a \neq x$, then the thesis is $\Gamma \vdash a : \tau$ which comes from the hypothesis by (2).

(SUBSUMPTION) The hypothesis comes from $\Gamma, x : \rho \vdash a : \tau'$ and $\tau' \leq \tau$. Then by induction we have $\Gamma \vdash a\{^{n_{[\varphi]}}/_x\} : \tau'$ and we conclude $\Gamma \vdash a\{^{n_{[\varphi]}}/_x\} : \tau$ by (SUBSUMPTION).

(DELIVERY) In this case $a \neq x$, then the thesis is $\Gamma \vdash a : \tau$, which comes from (2).

Point 2 follows by induction on the derivation of $\Gamma, x : \rho \vdash A$ and a case analysis on the last rule applied. We only show here the interesting cases.

(INPUT) Then $A = a(y_1 : \tau_1, \ldots, y_k : \tau_k).A_1$ and the hypothesis comes from $\Gamma, x : \rho, y_1 : \tau_1, \ldots, y_k : \tau_k \vdash A_1$ and $\Gamma, x : \rho \vdash a : \mathsf{G}[(\tau_1, \ldots, \tau_k)^{\mathsf{r}} \parallel \Delta]$. By induction we have $\Gamma, y_1 : \tau_1, \ldots, y_k : \tau_k \vdash A_1\{^{n_{[\varphi]}}/_x\}$ and $\Gamma \vdash a\{^{n_{[\varphi]}}/_x\} : \mathsf{G}[(\tau_1, \ldots, \tau_k)^{\mathsf{r}} \parallel \Delta]$, thus $\Gamma \vdash A\{^{n_{[\varphi]}}/_x\}$ by the rule (INPUT).

(OUTPUT) Then $A = \bar{a}\langle b_1, \ldots, b_k \rangle.A_1$ and the premises are $\Gamma, x : \rho \vdash a : \mathsf{G}[(\tau_1, \ldots, \tau_k)^{\mathsf{w}} \parallel \Delta]$, $\Gamma, x : \rho \vdash A_1$, $\Gamma, x : \rho \vdash b_i : \mathsf{G}_i[T_i \parallel \Delta_i]$ and $\mathsf{G} : T_i' \to \Delta_i' \in \Delta_i$ or $\mathsf{any} : T_i' \to \Delta_i' \in \Delta_i$ with $\mathsf{G}_i[T_i' \parallel \Delta_i'] \leq \tau_i$.

By induction we have $\Gamma \vdash a\{^{n[\varphi]}/_x\} : \mathsf{G}[(\tau_1,\ldots,\tau_k)^{\mathsf{w}} \parallel \Delta]$, $\Gamma \vdash A_1\{^{n[\varphi]}/_x\}$, $\Gamma \vdash b_i\{^{n[\varphi]}/_x\}$: $\mathsf{G}_i[T_i \parallel \Delta_i]$. And we conclude $\Gamma \vdash A\{^{n[\varphi]}/_x\}$ by applying the rule (OUTPUT).

(NEW m) Then $A = (\nu m : \tau)A_1$ and the hypothesis comes from $\Gamma, x : \rho, m : \tau \vdash A_1$ and $\Gamma, x : \rho \vdash \tau$. By induction we have $\Gamma, m : \tau \vdash A_1\{^{n[\psi]}/_x\}$ and we conclude by (NEW m).

□

5.2. *Main Theorems*

In the proofs below, in particular in the proof of Theorem 5.2, we make explicit appeal to a transitivity property of the subtype relation. As we noticed, our presentation of subtyping does *not* include a general rule of transitivity: on the other hand, as we show in Section 6, the form of transitivity we rely upon in the proofs of this section does indeed hold for the typing system. Also, we refer to the following, standard, notion of evaluation (or *dynamic*), single-hole context:

$$C \;::=\; - \;\mid\; (\nu n : \tau)C \;\mid\; (\nu \mathsf{G})C \;\mid\; C|A$$

and let $C[A]$ note the capture-free substitution of the closed dynamic process A for the hole '$-'$' in C.

Theorem 5.1 (Access Control). Let $A \equiv C[\overline{n_{[\varphi]}}\langle a_1,\ldots,a_k\rangle.B]$ be a closed well-typed dynamic process. Assume that $\Gamma \vdash \overline{n_{[\varphi]}}\langle a_1,\ldots,a_k\rangle.B$ for a suitable Γ. Then $\Gamma \vdash n_{[\varphi]} : \mathsf{G}[(\tau_1,\ldots,\tau_k)^{\mathsf{w}}]$ and for all $i = 1,...,k$, one has $\Gamma \vdash a_i : \sigma_i$, $\sigma_i{\downarrow}\mathsf{G}$ is defined, and $\sigma_i{\downarrow}\mathsf{G} \le \tau_i$. Similarly, if $A \equiv C[n_{[\varphi']}(x_1{:}\rho_1,\ldots,x_l{:}\rho_l).B]$ with $\Gamma \vdash n_{[\varphi']}(x_1{:}\rho_1,\ldots,x_l{:}\rho_l).B$, we have $\Gamma \vdash n_{[\varphi']} : \mathsf{G}[(\rho_1,\ldots,\rho_l)^{\mathsf{r}}]$.

Proof. By an inspection of the typing rules. □

Theorem 5.2 (Flow Control). Let $\Gamma' \vdash A$ with A closed. Assume, further, that $\Gamma' \vdash A$ depends on the judgment $\Gamma \vdash n_{[\varphi]} : \tau$ with $\varphi \neq \varepsilon$ (i.e. $\Gamma \vdash n_{[\varphi]} : \tau$ occurs in the derivation that proves $\Gamma' \vdash A$). Then $n \in Dom(\Gamma)$ and $\Gamma[\varphi] \neq \bot$. In addition, $\Gamma(n) = \rho$ with ρ such that $\rho{\downarrow}\Gamma[\varphi]$ is defined and $\rho{\downarrow}\Gamma[\varphi] \le \tau$.

Proof. By induction on the length of φ.

case $\varphi = m$. Then the hypothesis $\Gamma \vdash n_{[m]} : \tau$ comes by several (SUBSUMPTION) rules from $\Gamma \vdash n_{[m]} : \tau'$ and $\tau' \le \tau$, and an application of (DELIVERY) from $\Gamma \vdash n : \mathsf{G}[T_1 \parallel \Delta_1]$, $\Gamma \vdash m : \mathsf{G}_m[T_m \parallel \Delta_m]$ and $\mathsf{G}_m : T' \to \Delta' \in \Delta_1$ or $\mathsf{any} : T' \to \Delta' \in \Delta_1$ with $\tau' = \mathsf{G}[T' \parallel \Delta']$. The judgment $\Gamma \vdash n : \mathsf{G}[T_1 \parallel \Delta_1]$ must have come by (SUBSUMPTION) and (PROJECT) from $\Gamma(n) = \mathsf{G}[T \parallel \Delta] = \rho$ and $\mathsf{G}[T \parallel \Delta] \le \mathsf{G}[T_1 \parallel \Delta_1]$, which implies $\Delta \le \Delta_1$. Note that the thesis we want to prove is $\rho \downarrow \mathsf{G}_m \le \tau$. Let us distinguish two cases:

— $\mathsf{G}_m : T' \to \Delta' \in \Delta_1$ and $\tau' = \mathsf{G}[T' \parallel \Delta']$. From $\Delta \le \Delta_1$ we have two sub-cases:
  - $\mathsf{G}_m : T'' \to \Delta'' \in \Delta$ and $\mathsf{G}[T'' \parallel \Delta''] \le \tau'$. In this case $\rho \downarrow \mathsf{G}_m = \mathsf{G}[T'' \parallel \Delta'']$, then from $\mathsf{G}[T'' \parallel \Delta''] \le \tau'$ and $\tau' \le \tau$, we conclude by transitivity.
  - $\mathsf{G}_m \notin Dom(\Delta)$ and $\mathsf{any} : T'' \to \Delta'' \in \Delta$ with $\mathsf{G}[T'' \parallel \Delta''] \le \tau'$. In this case $\rho \downarrow \mathsf{G}_m = \mathsf{G}[T'' \parallel \Delta'']$, then from $\mathsf{G}[T'' \parallel \Delta''] \le \tau'$ and $\tau' \le \tau$, we conclude by transitivity.

— $\mathsf{any} : T' \to \Delta' \in \Delta_1$ and $\tau' = \mathsf{G}[T' \parallel \Delta']$. with $\mathsf{G}_m \notin Dom(\Delta_1)$. From $\Delta \le \Delta_1$ we have $\mathsf{any} : T'' \to \Delta'' \in \Delta$ with $\mathsf{G}[T'' \parallel \Delta''] \le \tau'$. We have two sub-cases:

- $\mathsf{G}_m : T'' \to \Delta'' \in \Delta$ Then $\rho \downarrow \mathsf{G}_m = \mathsf{G}[T'' \parallel \Delta'']$, and from $\Delta \leq \Delta_1$ (rule (SUB-Δ-3)) we have $\mathsf{G}[T'' \parallel \Delta''] \leq \mathsf{G}[T' \to \Delta'] = \tau'$. Then we conclude by transitivity since $\tau' \leq \tau$.
- $\mathsf{G}_m \notin \mathit{Dom}(\Delta)$. In this case $\rho \downarrow \mathsf{G}_m = \mathsf{G}[T'' \parallel \Delta'']$, then from $\mathsf{G}[T'' \parallel \Delta''] \leq \tau'$ and $\tau' \leq \tau$, we conclude by transitivity.

case $\varphi = \bar{\varphi}{::}m$. In this case the hypothesis $\Gamma \vdash n_{[\varphi]} : \tau$ comes by a number of (SUBSUMPTION) from $\Gamma \vdash n_{[\varphi]} : \tau'$ and $\tau' \leq \tau$, and an application of (DELIVERY) from $\Gamma \vdash n_{[\bar{\varphi}]} : \mathsf{G}[T_1 \parallel \Delta_1]$, $\Gamma \vdash m : \mathsf{G}_m[T_m \parallel \Delta_m]$ and $\mathsf{G}_m : T' \to \Delta' \in \Delta_1$ or $\mathsf{any} : T' \to \Delta' \in \Delta_1$ with $\tau' = \mathsf{G}[T' \parallel \Delta']$. From $\Gamma \vdash n_{[\bar{\varphi}]} : \mathsf{G}[T_1 \parallel \Delta_1]$, by induction, we have $\Gamma(n) = \mathsf{G}[T \parallel \Delta] = \rho$ and $\rho \downarrow \Gamma[\bar{\varphi}] \leq \mathsf{G}[T_1 \parallel \Delta_1]$. Then $\rho \downarrow \Gamma[\bar{\varphi}] = \mathsf{G}[T^* \parallel \Delta^*]$ and $\Delta^* \leq \Delta_1$. Let us distinguish two cases:

— $\mathsf{G}_m : T' \to \Delta' \in \Delta_1$ and $\tau' = \mathsf{G}[T' \parallel \Delta']$. From $\Delta^* \leq \Delta_1$ we have two sub-cases:

- $\mathsf{G}_m : T'' \to \Delta'' \in \Delta^*$ and $\mathsf{G}[T'' \parallel \Delta''] \leq \tau'$. In this case $\rho \downarrow \Gamma[\bar{\varphi}{::}m] = \mathsf{G}[T'' \parallel \Delta'']$, then from $\mathsf{G}[T'' \parallel \Delta''] \leq \tau'$ and $\tau' \leq \tau$, we conclude by transitivity.
- $\mathsf{G}_m \notin \mathit{Dom}(\Delta^*)$ and $\mathsf{any} : T'' \to \Delta'' \in \Delta^*$ with $\mathsf{G}[T'' \parallel \Delta''] \leq \tau'$. In this case $\rho \downarrow \Gamma[\bar{\varphi}{::}m] = \mathsf{G}[T'' \parallel \Delta'']$, then from $\mathsf{G}[T'' \parallel \Delta''] \leq \tau'$ and $\tau' \leq \tau$, we conclude by transitivity.

— $\mathsf{any} : T' \to \Delta' \in \Delta_1$ and $\tau' = \mathsf{G}[T' \parallel \Delta']$. with $\mathsf{G}_m \notin \mathit{Dom}(\Delta_1)$. From $\Delta^* \leq \Delta_1$ we have $\mathsf{any} : T'' \to \Delta'' \in \Delta^*$ with $\mathsf{G}[T'' \parallel \Delta''] \leq \tau'$. We have two sub-cases:

- $\mathsf{G}_m : T'' \to \Delta'' \in \Delta^*$ Then $\rho \downarrow \Gamma[\bar{\varphi}{::}m] = \mathsf{G}[T'' \parallel \Delta'']$, and from $\Delta^* \leq \Delta_1$ (rule (SUB-Δ-3)) we have $\mathsf{G}[T'' \parallel \Delta''] \leq \mathsf{G}[T' \to \Delta'] = \tau'$. Then we conclude by transitivity since $\tau' \leq \tau$.
- $\mathsf{G}_m \notin \mathit{Dom}(\Delta')$. In this case $\rho \downarrow \Gamma[\bar{\varphi}{::}m] = \mathsf{G}[T'' \parallel \Delta'']$, then from $\mathsf{G}[T'' \parallel \Delta''] \leq \tau'$ and $\tau' \leq \tau$, we conclude by transitivity.

□

Theorem 5.3 (Subject Reduction). If $\Gamma \vdash A$ and $A \longrightarrow^* B$, then $\Gamma \vdash B$.

Proof. By induction on the derivation of $A \longrightarrow B$ and a case analysis on the last rule applied.

— $A = \overline{n_{[\varphi]}}\langle m_{1[\varphi_1]}, \ldots, m_{k[\varphi_k]}\rangle.A_1 \mid n_{[\psi]}(x_1 : \tau_1, \ldots, x_k : \tau_k).A_2$. From $\Gamma \vdash A$ we have

$$
\begin{array}{ll}
\Gamma \vdash n_{[\varphi]} : \mathsf{G}'[(\tau'_1, \ldots, \tau'_k)^{\mathsf{w}} \parallel \Delta'] & (1)\\
\Gamma \vdash A_1 & (2)\\
\Gamma \vdash m_{i[\varphi_i]} : \mathsf{G}_i[T_i \parallel \Delta_i] \quad i = 1,..,k & (3)\\
\Delta_i(\mathsf{G}') = T'_i \to \Delta'_i \quad \mathsf{G}_i[T'_i \parallel \Delta'_i] = \tau'_i & (4)\\
\Gamma \vdash n_{[\psi]} : \mathsf{G}[(\tau_1, \ldots, \tau_k)^{\mathsf{r}} \parallel \Delta] & (5)\\
\Gamma, x_1 : \tau_1, \ldots, x_k : \tau_k \vdash A_2 & (6)
\end{array}
$$

The equalities in condition (4) correspond to the equivalent conditions $\mathsf{G}_i[T_i \parallel \Delta_i] \downarrow \mathsf{G}' = \tau'_i$ required in the typing of the output prefix of the process in question. From (1) and (5), by Lemma 5.2 we have $\Gamma \vdash n : \mathsf{G}[T \parallel \Delta]$, $\mathsf{G}' = \mathsf{G}$, $T \leq (\tau'_1, \ldots, \tau'_k)^{\mathsf{w}}$ and $T \leq (\tau_1, \ldots, \tau_k)^{\mathsf{r}}$. The last two judgments imply $\tau'_i \leq \tau_i$, $i = 1,..,k$. Now, from (3), $\Gamma \vdash n : \mathsf{G}[T \parallel \Delta]$ and (4), by (DELIVERY) we have $\Gamma \vdash m_{i[\varphi_i::n]} : \tau'_i$; from this judgement and $\tau'_i \leq \tau_i$, by (SUBSUMPTION), we derive $\Gamma \vdash m_{i[\varphi_i::n]} : \tau_i$, $i = 1,..,k$. The last judgments, together with (6), by the Substitution Lemma give $\Gamma \vdash A_2\{{}^{m_{i[\varphi_i::n]}}/_{x_i}\}$, that together with (2) by (PAR), give $\Gamma \vdash B$ as desired.

— In case $A = (\nu n{:}\tau)A_1 \longrightarrow B = (\nu n{:}\tau)B_1$ since $A_1 \longrightarrow B_1$. From $\Gamma \vdash A$ we have $\Gamma, n : \tau \vdash A_1$ and $\Gamma \vdash \tau$. By induction we also have $\Gamma, n : \tau \vdash B_1$, and we conclude $\Gamma \vdash B$ by an application of (NEW n).

— In case $A = (\nu \mathsf{G})A_1 \longrightarrow B = (\nu \mathsf{G})B_1$ since $A_1 \longrightarrow B_1$. From $\Gamma \vdash A$ we have $\Gamma, \mathsf{G} \vdash A_1$. By induction we also have $\Gamma, \mathsf{G} \vdash B_1$, and we conclude $\Gamma \vdash B$ by an application of (NEW G).

— In case $A = A_1 \mid A_2 \longrightarrow B = B_1 \mid A_2$ since $A_1 \longrightarrow B_1$. From $\Gamma \vdash A$ we have $\Gamma \vdash A_1$ and $\Gamma \vdash A_2$. By induction we also have $\Gamma \vdash B_1$ and we conclude $\Gamma \vdash B$ by an application of (PAR) from $\Gamma \vdash B_1$ and $\Gamma \vdash A_2$.

— In case $A \equiv A_1, A_1 \longrightarrow B_1$ and $B_1 \equiv B$. From $\Gamma \vdash A$, by Subject Congruence, we have $\Gamma \vdash A_1$, then by induction $\Gamma \vdash B_1$ and again by Subject Congruence $\Gamma \vdash B$ as desired.

□

We conclude the proof of type safety by projecting the results proved in Theorems 5.1 and 5.2 on static processes, and then on systems.

Corollary 5.1 (Safety). Let S be a closed system, and assume $\Gamma \vdash S$ for some Γ. Then, for all A such that $S \longrightarrow^* A$, one has $\Gamma \vdash A$ and both Theorems 5.1 and 5.2 hold of A.

Proof. From $\Gamma \vdash S$ and the format of the typing rule for principals, we know that $\Gamma \vdash_S [S]$. By Lemma Lemma 4.1, this implies $\Gamma \vdash [S]$, and then the proofs follow by Theorem 5.3.

We remark that while the safety theorem is stated for systems (equivalently, for static processes), Theorems 5.2 and 5.3 could not be meaningfully stated without appealing to the flow tags attached to names. In particular, it is not true that , $\Gamma \vdash A$ implies $\Gamma \vdash |A|$ for all dynamic processes A. To see that, note that $\Gamma \vdash A$ may depend on two tagged names $n_{[\varphi_1]}$ and $n_{[\varphi_2]}$ being given different types (not related by subtyping) by the (DELIVERY) rule. On the other hand, if we erase the tags, we lose the possibility of appealing to the (DELIVERY) rule, and consequently the judgment $\Gamma \vdash |A|$ fails. For this very reason, Theorem 5.3 does not hold, in general, under the reduction semantics $\mapsto$ of (Cardelli et al., 2005). Interestingly, however, we can recover subject reduction for $\mapsto$ provided that we make adequate assumptions on the structure of the types occurring in Γ and A. We formalize the relationship with the type system of (Cardelli et al., 2005), showing that our type system is a conservative extension of the type system of πG.

Given a πG type environment Γ and a πG process P, let $[\![\Gamma]\!]$ and $[\![P]\!]$ be the type environment and process that result from applying the encoding of types from Section 3 (on page 9) systematically to all types occurring in Γ and P.

Now call a type *simple* when it is the encoding of a πG type; similarly, call a type environment and a (dynamic) process *simple* when all the types occurring therein are simple. Then we have:

Theorem 5.4 (Relationships with πG). $\Gamma \vdash P$ is derivable in πG iff $[\![\Gamma]\!] \vdash_S [\![P]\!]$ is derivable in our type system.

Proof. (*Sketch*) The proof follows by observing that if we restrict to simple types, then the type formation rules as well as the (OUTPUT) rule for processes coincide with the corresponding rules in πG. Now, if Γ and A are simple, it is not difficult to see that $\Gamma \vdash A$ implies $\Gamma \vdash |A|$. Intuitively, the reason is that simple types are insensitive to flows: this is a consequence of the delivery type being the same at all hops in a simple type. □

Table 6 Sub-typing Procedure.

```
Sub-typing($\rho,\sigma$)
  define $J = \{\rho \leq \sigma\}$ and $H = H' = I = I' = J' = \emptyset$;
  while $H \cup I \cup J \neq \emptyset$ do
    if $J \neq \emptyset$ pick $\tau \leq \tau' \in J$ and apply one of the following rules:
    1.  if $\tau \leq \tau' \in J'$ then $J = J \setminus \{\tau \leq \tau'\}$;
        /* as $\tau \leq \tau'$ has been already treated */
    2.  if $\tau = \tau'$ then $J = J \setminus \{\tau \leq \tau'\}$;
        /* according to ($\tau$-REFLEX) */
    3.  otherwise let $\tau = \mathsf{G}[T \parallel \Delta]$ and $\tau' = \mathsf{G}'[T' \parallel \Delta']$
          if $\mathsf{G} \neq \mathsf{G}'$ then return false
          else $J = J \setminus \{\tau \leq \tau'\}$; $J' = J' \cup \{\tau \leq \tau'\}$; $I = I \cup \{T \leq T'\}$; $H = H \cup \{\Delta \leq \Delta'\}$;
        /* according to ($\tau$-TYPE/POLICY) */
    if $I \neq \emptyset$ pick $T \leq T' \in I$ and apply one of the following rules:
    1.  if $T \leq T' \in I'$ then $I = I \setminus \{T \leq T'\}$;
        /* as $T \leq T'$ has been already treated */
    2.  if $T = T'$ then $I = I \setminus \{T \leq T'\}$;
        /* according to ($T$-REFLEX) */
    3.  otherwise, let $T = (\tau_1 \dots \tau_n)^{\nu}$ and $T' = (\tau'_1 \dots \tau'_n)^{\nu'}$
          if $\nu \not\leq \nu'$ or $n \neq m$ then return false
          else   $I = I \setminus \{T \leq T'\}$; $I' = I' \cup \{T \leq T'\}$;
                 switch $\nu'$   case r:    $J = J \cup \{\tau_i \leq \tau'_i\}_{i=1 \dots n}$;
                                 case w:    $J = J \cup \{\tau'_i \leq \tau_i\}_{i=1 \dots n}$;
                                 case rw:   if $\tau_i \neq \tau'_i$ for some $i$ then return false;
        /* according to the structural rules */
    if $H \neq \emptyset$ pick $\Delta \leq \Delta' \in H$ and apply one of the following rules:
    1.  if $\Delta \leq \Delta' \in H'$ then $H = H \setminus \{\Delta \leq \Delta'\}$;
        /* according to ($\Delta$-ASSUME) */
    2.  if $\Delta = \Delta'$ then $H = H \setminus \{\Delta \leq \Delta'\}$;
        /* according to ($\Delta$-REFLEX) */
    3.  otherwise $H = H \setminus \{\Delta \leq \Delta'\}$, $H' = H' \cup \{\Delta \leq \Delta'\}$ and
        a. if $\Delta = \mu X.\Delta_1\{X\}$ then $H = H \cup \{\Delta_1\{\mu X.\Delta_1\{X\}\} \leq \Delta'\}$;
           /* according to ($\Delta$-LEFT-UNFOLD) */
        b. if $\Delta = \sum_i \mathsf{G}_i : T_i \to \Delta_i$ and $\Delta' = \mu X.\Delta'_1\{X\}$
           then $H = H \cup \{\Delta \leq \Delta'_1\{\mu X.\Delta'_1\{X\}\}\}$;
           /* according to ($\Delta$-RIGHT-UNFOLD) */
        c. otherwise call PolicyExt($\Delta,\Delta',H,H',I,I'$)
  return true
```

Based on this result, we immediately obtain subject reduction for simple processes.

Theorem 5.5 (Subject Reduction for simple processes). Assume $\Gamma \vdash_S P$ with Γ simple, and P closed and simple, and let $P \mapsto^* Q$. Then $\Gamma \vdash_S Q$.

Now the secrecy theorem of (Cardelli et al., 2005) can be re-established in our system with no additional effort for simple processes.

6. Decidability of typing and subtyping

As we noted earlier, our presentation of the subtype relation does not include a transitivity rule. In this section we show *(i)* that this make it possible to prove that the subtype relation is decidable,

Table 7 Policy-extension Procedure.

```
PolicyExt($\Delta,\Delta',H,I,H',I'$)
A.   if $\Delta=\sum_{i\in I\cup J}\mathsf{G}_i:T_i\rightarrow\Delta_i$ and $\Delta'=\sum_{i\in I}\mathsf{G}_i:T_i'\rightarrow\Delta_i'$ with $\mathsf{any}\notin\{\mathsf{G}_i\}_{i\in I\cup J}$
     then $H=H\cup\{\Delta_i\leq\Delta_i'\}_{i\in I}$ and $I=I\cup\{T_i\leq T_i'\}_{i\in I}$;
     /* according to ($\textsc{Sub-}\Delta\text{-}1$) */
B.   if $\Delta=\Delta_1+\mathsf{any}:T_d\rightarrow\Delta_d$ and $\Delta'=\Delta_1'+\sum_{i\in I}\mathsf{G}_i:T_i\rightarrow\Delta_i$
     with $\mathsf{any}\notin Dom(\Delta_1')\cup\{\mathsf{G}_i\}_{i\in I}$ and $Dom(\Delta_1')\subseteq Dom(\Delta_1)$ and $\{\mathsf{G}_i\}_{i\in I}\cap Dom\Delta_1=\emptyset$
     then $H=H\cup\{\Delta_1\leq\Delta_1'\}\cup\{\Delta_d\leq\Delta_i'\}_{i\in I}$ and $I=I\cup\{T_d\leq T_i\}_{i=\in I}$
     /* according to ($\textsc{Sub-}\Delta\text{-}2$) */
C.   if $\Delta=\Delta_1+\mathsf{any}:T_d\rightarrow\Delta_d+\sum_{i\in I}\mathsf{G}_i:T_i\rightarrow\Delta_i$ and $\Delta'=\Delta_1'+\mathsf{any}:T_d'\rightarrow\Delta_d'$
     with $Dom(\Delta_1)\subseteq Dom(\Delta_1')$ and $\{\mathsf{G}_i\}_{i\in I}\cap Dom(\Delta_1')=\emptyset$
     then $H=H\cup\{\Delta_1+\mathsf{any}:T_d\rightarrow\Delta_d\leq\Delta_1'\}\cup\{\Delta_d\leq\Delta_d'\}\cup\{\Delta_i\leq\Delta_d'\}_{i=1\ldots n}$
     and $I=I\cup\{T_d\leq T_d'\}\cup\{T_i\leq T_d'\}_{i\in I}$;
     /* according to ($\textsc{Sub-}\Delta\text{-}3$) */
D.   otherwise return **false**
     /* as $\mathsf{any}$ appears just in $\Delta'$ and not in $\Delta$ */
```

and *(ii)* that the form of transitivity used in the proof of Theorem 5.2 is still admissible in the typing system. By the decidability of subtyping, we then derive that typing is decidable as well.

We start by introducing the procedure `Sub-typing`(ρ,σ) outlined in Table 6, which decides the subtype relation. The procedure implements a bottom-up search for a derivation rooted at $\vdash\rho\leq\sigma$: at every step, it first tries to apply the rules ($\textsc{Reflex}$) and (Δ-$\textsc{Assume}$) and in case it fails, it looks for a possible rule according to the structure of the types or the delivery policies. In particular, in case of resource types $\tau_1\leq\tau_2$ and structural types $T_1\leq T_2$, the procedure checks whether the inequality was already treated, and, in that case, it does not analyze the pair further. In fact, since recursion occurs in delivery policies, $\tau_1\leq\tau_2$ and $T_1\leq T_2$ can only appear in the conclusion of the derivation for $\vdash\tau_1\leq\tau_2$ and $\vdash T_1\leq T_2$, respectively.

In detail, the procedure builds a triple of pairs of sets (H,H'), (I,I') and (J,J'). The sets (H,H') keep track of all the pairs $\Delta_1\leq\Delta_2$ generated and processed respectively, the sets (I,I') keep track of all the pairs $T_1\leq T_2$, and finally the sets (J,J') deal with the pairs $\tau_1\leq\tau_2$. Collectively, these sets make it possible to control the application of the rules numbered `3` in Table 6 – these are the only rules which extend the sets H,I and J – and in particular to ensure that such rules process every pair appearing in $H\cup I\cup J$ at most once during the computation. If none of the rules can be applied and $H\cup\ I\cup J\neq\emptyset$ then the procedure fails, otherwise it succeeds.

6.1. *Termination of the subtyping algorithm*

To show termination, we have to count the pairs that appear in $H\cup I\cup J$ during the computation and to prove that there are finitely many of them. Basically, we need to show that a type, and a recursive policy in particular, has a finite number of sub-expressions. As noted in (Gapeyev et al., 2002), proving this, seemingly obvious, property requires a quite some of work. The difficulty is that there are two possible ways of defining the set of 'closed sub-expressions' of a type. One, the *top-down* sub-expression $\sqsubseteq_{td}$, corresponds directly to the sub-expressions generated by the procedure `Sub-typing`, and by the rules in Table 4. The other, defined *bottom-up* sub-expression $\sqsubseteq_{bu}$, supports a straightforward proof that the set of closed sub-expressions of every closed type is finite. The two relations are defined on the expressions $e::=\tau\mid T\mid\Delta$ by the following rules:

Common rules:

$$\frac{}{\tau \sqsubseteq \tau} \qquad \frac{}{T \sqsubseteq T} \qquad \frac{}{\Delta \sqsubseteq \Delta} \qquad \frac{e \sqsubseteq T}{e \sqsubseteq \mathsf{G}[T \parallel \Delta]} \qquad \frac{e \sqsubseteq \Delta}{e \sqsubseteq \mathsf{G}[T \parallel \Delta]}$$

$$\frac{e \sqsubseteq \tau_i}{e \sqsubseteq (\tau_1 \ldots \tau_n)^{\mathsf{v}}}\ (\text{i=1...n}) \qquad \frac{e \sqsubseteq T}{e \sqsubseteq \mathsf{G} : T \to \Delta} \qquad \frac{e \sqsubseteq \Delta}{e \sqsubseteq \mathsf{G} : T \to \Delta} \qquad \frac{e \sqsubseteq \Delta}{e \sqsubseteq \Delta + \Delta'}$$

Specific rules:

$$\frac{e \sqsubseteq_{td} \Delta\{\mu X.\Delta\{X\}\}}{e \sqsubseteq_{td} \mu X.\Delta\{X\}} \qquad \frac{e\{X\} \sqsubseteq_{bu} \Delta\{X\}}{e\{\mu X.\Delta\{X\}\} \sqsubseteq_{bu} \mu X.\Delta\{X\}}$$

The proof of termination proceeds by showing that $\sqsubseteq_{td}$ is a subset of $\sqsubseteq_{bu}$, as done in (Brandt and Henglein, 1998). We start with the proof of some basic properties. First the transitivity of $\sqsubseteq_{td}$.

Lemma 6.1. If $e_1 \sqsubseteq_{td} e_2$ and $e_2 \sqsubseteq_{td} e_3$, then $e_1 \sqsubseteq_{td} e_3$.

Proof. By induction on the derivation of $e_2 \sqsubseteq_{td} e_3$. □

Moreover, the number of bottom-up sub-expressions is finite.

Lemma 6.2. The set $\{e' : e' \sqsubseteq_{bu} e\}$ is finite for each e.

Proof. Straightforward structural induction on e, using the following observations, that are consequences of the definition of $\sqsubseteq_{bu}$.

— if $e = X$, then $\{e' : e' \sqsubseteq_{bu} e\} = \{X\}$;

— if $e = \mu X.\Delta_1\{X\}$, then $\{e' : e' \sqsubseteq_{bu} e\} = \{e\} \cup \{e'\{e\} : e'\{X\} \sqsubseteq_{bu} \Delta_1\{X\}\}$;

— $e = \mathsf{G}[T \parallel \Delta]$ with $T = (\tau_1, \ldots, \tau_n)^{\mathsf{v}}$, then $\{e' : e' \sqsubseteq_{bu} e\} = \{e\} \cup \{e' : e' \sqsubseteq_{bu} \tau_i\}_{i=1\ldots n} \cup \{e' : e' \sqsubseteq_{bu} \Delta(\mathsf{G}')\}_{\mathsf{G}' \in Dom(\Delta)}$. □

As done in (Gapeyev et al., 2002), to prove that the bottom-up sub-expressions of a type include its top-down sub-expressions, we need the following lemma, which relates bottom-up sub-expressions and substitution, and whose proof follows the lines of (Gapeyev et al., 2002).

Lemma 6.3. For every expression $e\{X\}$, if $e' \sqsubseteq_{bu} e\{\Delta\}$ then either $e' \sqsubseteq_{bu} \Delta$ or $e' = e''\{\Delta\}$ for some $e''\{X\}$ with $e''\{X\} \sqsubseteq_{bu} e\{X\}$.

Proof. Structural induction on e, see (Gapeyev et al., 2002). □

We can now relate the two notions of sub-expressions.

Lemma 6.4. If $e \sqsubseteq_{td} e'$ then $e \sqsubseteq_{bu} e'$.

Proof. Proceed by induction on the derivation of $e' \sqsubseteq_{td} e$, and show that the rules for $\sqsubseteq_{td}$ are sound for $\sqsubseteq_{bu}$. The only interesting case is the rule

$$\frac{e \sqsubseteq_{td} \Delta\{\mu X.\Delta\{X\}\}}{e \sqsubseteq_{td} \mu X.\Delta\{X\}}$$

Assume that $e \sqsubseteq_{bu} \Delta\{\mu X.\Delta\{X\}\}$ then Lemma 6.3 says that either *(i)* $e \sqsubseteq_{bu} \mu X.\Delta\{X\}$ or *(ii)*

$e = e''\{\mu X.\Delta\{X\}\}$ with $e''\{X\} \sqsubseteq_{bu} \Delta\{X\}$. The case *(i)* gives the conclusion of the rule. In case *(ii)* conclude $e \sqsubseteq_{bu} \mu X.\Delta\{X\}$ by applying the rule

$$\frac{e''\{X\} \sqsubseteq_{bu} \Delta\{X\}}{e''\{\mu X.\Delta\{X\}\} \sqsubseteq_{bu} \mu X.\Delta\{X\}}$$

$\square$

Lemma 6.5. The set $\{e : e \sqsubseteq_{td} \tau\}$ is finite for each τ.

Proof. Apply Lemma 6.2 and Lemma 6.4. $\square$

We are ready to prove the termination of the sub-typing procedure.

Theorem 6.1 (Termination). The subtyping procedure stops on every input.

Proof. The procedure may loop only if one of the rules marked by `3` applies infinitely many times. Since the procedure first checks H', I' and J' , rules numbered `3` may be invoked at most once on every pair that appears in $H \cup I \cup J$. The only way to extend the set $H \cup I \cup J$ is by one of the rules `3`. All the pairs generated by these rules are top-down sub-expressions of the current pair. As $H \cup I \cup J$ is initialized to the input pair $\{\rho \leq \sigma\}$, the transitivity of $\sqsubseteq_{td}$ (cf. Lemma 6.1) says that all the elements which may possibly appear in $H \cup I \cup J$ are in $\{e \leq e' : e \sqsubseteq_{td} \sigma \text{ and } e' \sqsubseteq_{td} \rho\}$, which is a finite set by Lemma 6.5. Then rules `3` may be applied a finite number of times. Hence the procedure always stops either with success or failure. $\square$

6.2. *Soundness and completeness of the subtyping algorithm*

We proceed with our analysis by showing the algorithm correct. The soundness part of the proof is straightforward.

Theorem 6.2 (Soundness). If the call `Sub-typing`(ρ,σ) succeeds, then there exists a derivation for $\vdash \rho \leq \sigma$.

Proof. The procedure `Sub-typing`(ρ,σ) builds a derivation for $\vdash \rho \leq \sigma$ according to the rules in Table 4: every pair $H \cup I \cup J$, H' that appears in any iteration can be interpreted as the set of sub-typing judgments $\{H' \vdash e \leq e' : e \leq e' \in H \cup I \cup J\}$; the procedure returns **`true`** when it reaches the axioms of the calculus. $\square$

The proof of completeness is more elaborate. It draws on, and extends, a corresponding proof for the system of I/O-types in (Pierce and Sangiorgi, 1996). As in that case, an inductive proof on derivations does not work, given the co-inductive nature of the presentation of the subtype relation (cf. the format of the rules for recursive subtyping and the presence of the (Δ-ASSUME) rule). To prove our claim, we therefore introduce a simulation relation over closed types and show that *(i)* it contains the subtype relation, and *(ii)* is decided by the `Sub-typing` algorithm. The two results are proved in Theorems 6.3 and 6.4, respectively. Together with Theorem 6.2, this proves that subtyping is decidable.

The definition of the simulation relation extends the corresponding definition found in (Pierce and Sangiorgi, 1996) by viewing our resource types as automata, or processes, performing transitions corresponding to the structure of the delivery policies they represent. The transitions on delivery policies are detailed below.

$$\frac{}{\Delta_1 + (G : T' \to \Delta') + \Delta_2 \xrightarrow{\mathsf{G}} (\Delta', T')} \qquad \frac{\Delta \xrightarrow{\mathsf{G}} (\Delta'\{X\}, T')}{\mu X.\Delta \xrightarrow{\mathsf{G}} (\Delta'\{\mu X.\Delta\}, T')}$$

Two remarks are in order. Firstly, due to the definition of Δ, this transition system is deterministic, namely the set $\mathit{reach}_{\mathsf{G}}(\Delta) \triangleq \{(\Delta', T') : \Delta \xrightarrow{\mathsf{G}} (\Delta', T')\}$ is either a singleton $\{(\Delta', T')\}$ or the empty set. Secondly, it is easy to verify that the labelled transition system respects type equality, as stated by the following lemma.

Lemma 6.6. For every group name G, we have $\mathit{reach}_{\mathsf{G}}(\mu X.\Delta\{X\}) = \mathit{reach}_{\mathsf{G}}(\Delta\{\mu X.\Delta\{X\}\})$.

In defining the simulation relation, we find it convenient to introduce additional notation.

$$\mathit{act}(\Delta) \triangleq \{\mathsf{G} : \mathit{reach}_{\mathsf{G}}(\Delta) \neq \emptyset\}$$

Item 1 of the definition below relates only types with the same group and compatible policies. Item 2 says that type simulation extends the tree sub-sort relation of (Pierce and Sangiorgi, 1996), which deals only with structural and recursive types. Items 3 is inspired by the definition of simulation on labelled transitions systems.

Definition 6.1 (Type Simulation). Let R_1 a binary relation on types, R_2 a binary relation on structural types and R_3 a binary relation on Delivery policies. The triple $\mathcal{R} = (R_1, R_2, R_3)$ is a type simulation when:

1 for every $(\mathsf{G}[T \parallel \Delta], \mathsf{G}'[T' \parallel \Delta']) \in R_1$ we have $\mathsf{G} = \mathsf{G}'$, $(T, T') \in R_2$ and $(\Delta, \Delta') \in R_3$;

2 for every $(\,(\tau_1, \dots, \tau_m)^{\nu}, (\tau'_1, \dots, \tau'_n)^{\nu'}\,) \in R_2$ we have $n = m$, $\nu \leq \nu'$ and for every $i = 1, \dots, m$: *(i)* if $\nu' = \mathsf{w}$, then $(\tau_i, \tau'_i) \in R_1$; *(ii)* if $\nu' = \mathsf{r}$, then $(\tau'_i, \tau_i) \in R_1$; *(iii)* if $\nu' = \mathsf{rw}$, then $\tau_i = \tau'_i$.

3 for every $(\Delta, \Delta') \in R_3$ we have:

(a) if $\Delta \xrightarrow{\mathsf{G}} (\Delta_1, T)$ then $\Delta' \xrightarrow{\mathsf{G}} (\Delta'_1, T')$ with $(\Delta_1, \Delta'_1) \in R_3$ and $(T, T') \in R_2$, otherwise $\Delta' \xrightarrow{\mathsf{any}} (\Delta'_1, T')$ with $(\Delta_1, \Delta'_1) \in R_3$ and $(T, T') \in R_2$;

(b) if $\Delta \xrightarrow{\mathsf{any}} (\Delta_1, T)$ then for every $(\Delta'_1, T') \in \bigcup_{\mathsf{G} \in \mathit{act}(\Delta') \setminus \mathit{act}(\Delta)} \mathit{reach}_{\mathsf{G}}(\Delta')$ it holds $(\Delta_1, \Delta'_1) \in R_3$ and $(T, T') \in R_2$.

We say that the type ρ simulates the type σ, symbolically $\sigma \lesssim \rho$, if there exists a simulation $\mathcal{R} = (R_1, R_2, R_3)$ such that $(\sigma, \rho) \in R_1$. Likewise, we say that the policy Δ' simulates the policy Δ, symbolically $\Delta \lesssim \Delta'$, if there exists a simulation $\mathcal{R} = (R_1, R_2, R_3)$ such that $(\Delta, \Delta') \in R_3$.

Lemma 6.7. The relation $\lesssim$ on types and delivery policies is reflexive and transitive.

Proof. For reflexivity, consider the identity relations $\mathrm{Id}_\tau = \{(\tau, \tau) : \tau \text{ is a resource type }\}$, $\mathrm{Id}_T = \{(T, T) : T \text{ is a structural type }\}$ and $\mathrm{Id}_\Delta = \{(\Delta, \Delta) : \Delta \text{ is a delivery policy }\}$. Then $\mathbf{Id} = (\mathrm{Id}_\tau, \mathrm{Id}_T, \mathrm{Id}_\Delta)$ is a simulation. For transitivity show that if (R'_1, R'_2, R'_3) and (R''_1, R''_2, R''_3) are simulations then the triple of relations (R_1, R_2, R_3), defined as $(e', e'') \in R_i$ iff there exists e such that $(e', e) \in R'_i$ and $(e, e'') \in R''_i$ for $i = 1, 2, 3$, is a simulation as well. $\square$

Type simulation is well defined as it respects type equality. In fact, it is straightforward to see that it is preserved by renaming of bound variables and permutation of delivery constraints inside delivery policies. Moreover the following lemma shows that also the unfolding of recursive policies preserves type simulation.

Lemma 6.8. For every structural type T, group G, delivery policy Δ and type τ it holds:

1 $\mu X.\Delta\{X\} \lesssim \Delta'$ if and only if $\Delta\{\mu X.\mathsf{G}[T \parallel \Delta\{X\}]\} \lesssim \Delta'$,

2 $\Delta' \lesssim \Delta\{\mu X.\mathsf{G}[T \parallel \Delta\{X\}]\}$ if and only if $\Delta' \lesssim \mu X.\Delta\{X\}$.

Proof. For the forward direction of item 1, assume $(\mu X.\Delta\{X\},\Delta') \in R_3$, where (R_1,R_2,R_3) is a simulation, and observe that by extending $R_3 \cup \{(\Delta\{\mu X.\mathsf{G}[T \parallel \Delta\{X\}]\},\Delta')\}$ we obtain a simulation thanks to Lemma 6.6. The backward direction and item 2 are analogous. □

The next theorem shows that the subtyping relation over resource types is included in $\lesssim$, in the following sense.

Theorem 6.3 (Sub-typing and Simulation). If $\vdash \rho \leq \sigma$ is derivable, then $\sigma \lesssim \rho$.

Proof. Let Π be a derivation for $\vdash \sigma \leq \rho$ and define

$$\begin{array}{rcl} R_1^\Pi & \triangleq & \{(\tau',\tau) : \Sigma \vdash \tau \leq \tau' \text{ is a judgment in } \Pi\} \\ R_2^\Pi & \triangleq & \{(T',T) : \Sigma \vdash T \leq T' \text{ is a judgment in } \Pi\} \\ R_3^\Pi & \triangleq & \{(\Delta',\Delta) : \Sigma \vdash \Delta \leq \Delta' \text{ is a judgment in } \Pi\} \end{array}$$

We prove that $\mathcal{R}^\Pi = (R_1^\Pi,R_2^\Pi,R_3^\Pi) \cup \mathbf{Id}$ is a type simulation. Pick $(e',e) \in R_i^\Pi$ and check that the requirements of Definition 6.1 are met. We reason by cases on the last rule applied in the sub-derivation of Π which is rooted at $\Sigma \vdash e \leq e'$.

(REFLEX) Then $e = e'$, hence Definition 6.1 is satisfied as $\mathbf{Id} \subseteq \mathcal{R}^\Pi$.

(τ-TYPE/POLICY), (T-READ), (T-WRITE) and (T-READ/WRITE) Then Definition 6.1 is satisfied thanks to the premises of the rules.

(SUB-Δ-i) Again, Definition 6.1 is satisfied thanks to the premises of the rules.

(Δ-ASSUME) The last judgment is $\Sigma \vdash \Delta \leq \Delta'$ and $\Delta \leq \Delta' \subseteq \Sigma$. Note that the root of the derivation Π exhibits an empty sub-policy environment, hence the application of (Δ-ASSUME) must appear as a leaf of a sub-derivation rooted at $\Sigma' \vdash \Delta \leq \Delta'$, with $\Delta \leq \Delta' \notin \Sigma'$, that ends with one or two applications of the (UNFOLD) rules, depending on the structure of Δ and Δ'. Moreover, those rules must be applied just after an instance of a (SUB-Δ) rule. Now, the premises of (SUB-Δ) rules, along with Lemma 6.6, say that Definition 6.1 is satisfied also in this case.

(Δ-LEFTUNFOLD) Consider the rule applied in Π just above this one: it must be either an instance of one among (Δ-REFLEX), (Δ-ASSUME), (SUB-Δ) or an instance of the rule (Δ-RIGTHUNFOLD) that follows the application of (Δ-REFLEX) or (Δ-ASSUME) or (SUB-Δ). In any case apply again one of the previous cases and Lemma 6.6.

(Δ-RIGTHUNFOLD) Analogous to the previous case. □

Another important property is that the subtyping procedure succeeds on inputs that are in the simulation relation, as shown in the following theorem.

Theorem 6.4 (Simulation Success). If $\sigma \lesssim \rho$, then `Sub-typing`(ρ,σ) succeeds.

Proof. Assume $\sigma \lesssim \rho$ and run the procedure `Sub-typing`(ρ,σ). Theorem 6.1 says that the procedure stops. Moreover $\sigma \lesssim \rho$ implies that $e \lesssim e'$ for every pair $e' \leq e$ included in $H \cup J \cup I$. Hence the checks at the points `3` always fail. Thus the procedure stops with **`true`**. $\square$

Now, by merging the results of Theorems 6.3, 6.4 and 6.2, we obtain the following equivalences.

Theorem 6.5 (Subtyping, Simulation and Procedure). For every types ρ and σ, the following statements are equivalent:

1 The judgment $\vdash \rho \leq \sigma$ is derivable.

2 ρ simulates σ, namely $\sigma \lesssim \rho$.

3 The call `Sub-typing`(ρ,σ) returns **`true`**.

Corollary 6.1 (Decidability). The subtyping relation is decidable.

Proof. Theorem 6.1 says that the call `Sub-typing`(ρ,σ) stops independently of input, and Theorem 6.5 says that `Sub-typing`(ρ,σ) returns **`true`** if and only if $\vdash \rho \leq \sigma$. $\square$

6.3. *Transitivity of Subtyping*

As a corollary of the decidability proof, we also obtain a proof that the subtype relation is transitive. In particular, we show that the axiomatization of subtyping admits a weak, but still useful in our proofs, form of transitivity that only applies to subtype judgments with empty environment.

Corollary 6.2 (Transitivity). The rule

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

is sound in the sub-typing system.

Proof. Theorem 6.5 says that, when the environment is empty, $\leq$ is equivalent to $\lesssim$, and Lemma 6.7 says that $\lesssim$ is transitive. $\square$

The result does not generalize to arbitrary judgments. To illustrate, letting $\Sigma = \Delta_1 \leq \Delta_2, \Delta_2 \leq \Delta_3$, we can derive $\Sigma \vdash \Delta_1 \leq \Delta_2$ and $\Sigma \vdash \Delta_2 \leq \Delta_3$ by (Δ-ASSUME). On the other hand, we clearly cannot derive $\Sigma \vdash \Delta_1 \leq \Delta_3$ directly, unless we have additional information on the structure of the policies involved. This is in fact an instance of a general problem for coinductive systems that are known to not interact well with transitivity rules (Gapeyev et al., 2002). Nevertheless, the rule of transitivity of Corollary 6.2 is still useful for our proofs, as the use of subtype judgments in the typing system is only on judgments with empty environment.

6.4. *Algorithmic Typing*

A further consequence of the decidability proof for subtyping, is that type checking itself is decidable. The proof follows the standard pattern, namely we give an algorithmic (and decidable) version of the typing system, that dispenses with the (SUBSUMPTION) rule, and show that the algorithmic system is sound and complete.

Given that we are interested in static typing, it is enough to define the algorithmic version for the static typing system $\vdash_S$. That is good news, because in the static system we can safely disregard the tagged names, and hence the algorithmic system is easily derived from the static typing system, by dropping the (SUBSUMPTION) rule and by replacing the rules (INPUT) and (OUTPUT) with the two rules below.

(INPUT-A)

$$\frac{\Gamma \vdash a : \mathsf{G}[(\tau_1,\dots,\tau_k)^{\nu} \parallel \Delta] \quad \Gamma, x_1:\tau_1,\dots,x_k:\tau_k \vdash P \quad \nu \le \mathsf{r},\ 0 \le k}{\Gamma \vdash a(x_1:\tau_1,\dots,x_k:\tau_k).P}$$

(OUTPUT-A)

$$\frac{\Gamma \vdash a : \mathsf{G}[(\tau_1,\dots,\tau_k)^{\nu} \parallel \Delta], \nu \le \mathsf{w} \quad \Gamma \vdash P \quad \Gamma \vdash b_i : \widehat{\tau}_i \quad \widehat{\tau}_i \downarrow \mathsf{G} \le \tau_i \quad \forall i \in [0..k]}{\Gamma \vdash \overline{a}\langle b_1,\dots,b_k\rangle.P}$$

Let $\vdash_A$ denote the typing relation in the algorithmic typing system.

Theorem 6.6. The typing relation $\vdash_A$ is decidable.

Proof. If we measure the size of a typing judgement (of the three kinds $\Gamma \vdash_A P$, $\Gamma \vdash_A a : \tau$, $\Gamma_A \vdash \diamond$) in terms of the sizes of the component types and processes, it is easily seen that for each typing rule in the algorithmic system, all the typing judgements in the premises have strictly smaller sizes than the judgement in the conclusion. Also, observe that the typing rules are syntax-directed, and that the absence of the subsumption rule ensures that $\Gamma \vdash_A a : \tau$ iff $a : \tau \in \Gamma$. Hence. for each rule of the algorithmic system all the types that occur in the typing and subtyping premises are determined by the types occurring in the conclusion (either in the environment or in the subject of the judgement). From this, the theorem follows directly by the proof that subtyping is decidable. □

Based the previous theorem, to show that static typing is decidable it is enough to show that algorithmic typing is sound and complete. The soundness part of the proof follows directly by induction: we just have to observe that *(i)* (INPUT-A) is a special case of (INPUT), and that *(ii)* whenever an algorithmic derivation uses (OUTPUT-A), a corresponding non-algorithmic derivation exists that uses (OUTPUT) and (SUBSUMPTION): the use of subsumption may be required to promote the types b_i so as to ensure that the policy Δ_i satisfies the constraints $\widehat{\tau}_i \downarrow \mathsf{G} = \tau_i$.

Theorem 6.7 (Algorithmic typing is sound). $\Gamma \vdash_A P$ implies $\Gamma \vdash_S P$.

For the opposite direction (completeness), the proof is more elaborate, but still standard. We need two lemmas: the first is a weakening lemma about type derivations; the second relates the typing of names in the static and algorithmic systems. Let the height of a derivation be defined as follows: the height of a derivation rooted at a subtyping judgement or at a typing judgement of the form $\Gamma \vdash \diamond$ or $\Gamma \vdash a : \tau$ is zero. The height of a derivation ending up with a rule whose conclusion is the judgement $\Gamma \vdash P$ is 1 + the maximal height of the derivations rooted at the premises of the rule.

Lemma 6.9. Let $\Gamma, x : \tau \vdash_S J$ be a derivable judgement, where J is any of $a : \tau, \diamond$ or P. Then, for all τ' such that $\vdash \tau' \leq \tau$ and $\Gamma \vdash_S \tau'$, the judgement $\Gamma, x : \tau' \vdash_S J$ is derivable as well, with a derivation of the same height.

Proof. The hypothesis $\Gamma \vdash_S \tau'$ is needed for the environment formation rules, in particular, for (ENV x), as τ' might introduce group names not in Γ. The proof is by induction on the derivation of the judgement $\Gamma, x : \tau \vdash_S J$, and follows directly by the induction hypothesis. Notice, in particular, that the uses of subsumption that may be required to complete the derivation of $\Gamma, x : \tau' \vdash_S J$ do not increase the height of the derivation (because of our definition of height). $\square$

Lemma 6.10. If $\Gamma \vdash_S a : \tau$, then $\Gamma \vdash_A a : \tau'$ with $\vdash \tau' \leq \tau$.

Proof. By induction on the derivation of the judgement in the hypothesis. There are only two cases, in fact: when the derivation ends up with rule (PROJECT) and (SUBSUMPTION). In the first case $\tau' = \tau$, whereas the second case follows directly by the induction hypothesis. For the (PROJECT) case, observe that $\Gamma \vdash_S \diamond$ iff $\Gamma \vdash_A \diamond$ as the type and environment formation rules in the two systems are the same. $\square$

Theorem 6.8 (Algorithmic typing is complete). $\Gamma \vdash_S P$ implies $\Gamma \vdash_A P$.

Proof. By induction on the height of the derivation of $\Gamma \vdash_S P$, and a case analysis of the last rule in the derivation. The cases (NEW G), (NEW n), (PAR), and (REPL) follow directly by the induction hypothesis, while (DEAD) follows by the observation that $\Gamma \vdash_S \diamond$ implies $\Gamma \vdash_A \diamond$.

In case (INPUT), the judgement $\Gamma \vdash_S P$ must be of the form $\Gamma \vdash_S a(x_1 : \tau_1, \ldots, x_n : \tau_n).P$, derived from $\Gamma \vdash_S a : \mathsf{G}[(\tau_1, \ldots, \tau_n)^{\mathsf{r}} \parallel \Delta]$ and $\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_S P$: let h be the height of the derivation rooted at this last judgement. By Lemma 6.10 we know that $\Gamma \vdash_A a : \tau$ with $\tau \leq \mathsf{G}[(\tau_1, \ldots, \tau_n)^{\mathsf{r}} \parallel \Delta]$. This, in turn, implies that $\tau = \mathsf{G}[(\tau'_1, \ldots, \tau'_n)^{\nu} \parallel \Delta']$ with $\nu \leq \mathsf{r}$ and $\tau'_i \leq \tau_i$. By Lemma 6.9, it follows that $\Gamma, x_1 : \tau'_1, \ldots, x_n : \tau'_n \vdash_S P$ is derivable with a derivation of height h, Then, by the induction hypothesis $\Gamma, x_1 : \tau'_1, \ldots, x_n : \tau'_n \vdash_A P$ and the desired judgement is derived by an application of (INPUT-A).

In case (OUTPUT), the judgement must be of the form $\Gamma \vdash_S \overline{a}\langle b_1, \ldots, b_k \rangle.P$, derived from the judgements *(i)* $\Gamma \vdash_S P$, *(ii)* $\Gamma \vdash_S a : \mathsf{G}[(\tau_1, \ldots, \tau_k)^{\mathsf{w}} \parallel \Delta]$ and *(iii)* $\Gamma \vdash_S b_i : \mathsf{G}_i[T_i \parallel \Delta_i]$, under the additional constraints that $\Delta_i(\mathsf{G}) = \overline{T}_i \to \overline{\Delta}_i$ and $\mathsf{G}_i[\overline{T}_i \parallel \overline{\Delta}_i] = \tau_i$ for all $i \in [0..k]$.

From *(i)*, by the induction hypothesis we know that $\Gamma \vdash_A P$. From *(ii)*, by Lemma 6.10 it follows that $\Gamma \vdash_A a : \mathsf{G}[(\tau^A_i, \ldots, \tau^A_k)^{\nu} \parallel \Delta^A]$ with $\nu \leq \mathsf{w}$, $\vdash \Delta^A \leq \Delta$ and $\tau_i \leq \tau^A_i$. Similarly, from *(iii)*, again by Lemma 6.10 it follows that $\Gamma \vdash_A b_i : \mathsf{G}_i[T^A_i \parallel \Delta^A_i]$, with $T^A_i \leq T_i$ and $\vdash \Delta^A_i \leq \Delta_i$. Letting $\Delta^A_i(\mathsf{G}) = \overline{T}^A_i \to \overline{\Delta}^A_i$, to conclude we need to show that $\mathsf{G}_i[\overline{T}^A_i \parallel \overline{\Delta}^A_i] \leq \tau^A_i$. This, in turn follows from $\mathsf{G}_i[\overline{T}_i \parallel \overline{\Delta}_i] = \tau_i$ and $\tau_i \leq \tau^A_i$ because $\Delta^A_i \leq \Delta_i$ implies $\overline{T}^A_i \leq \overline{T}_i$ and $\overline{\Delta}^A_i \leq \overline{\Delta}_i$. $\square$

7. Conclusion

We have developed a type theory for the specification and the static analysis of access control policies in the pi-calculus. Our approach extends and complements previous work on the subject by introducing a new class of types so defined as to control the dynamic flow of values among system components. We have shown the flexibility of our systems with several examples, and proved that it provides strong safety guarantees for all well-typed processes.

Our present results are proved for systems that only include well-typed components. However, we believe it is possible to capture a stronger results, such as those known as *robust safety* that guarantee safety in the presence of an untyped opponent. Indeed, we do not see any fundamental impediment in adapting existing approaches to the problem (cf. e.g, (Gordon and Jeffrey, 2004; Fournet et al., 2007), and (Cardelli et al., 2005) for that matter). As the secrecy result in (Cardelli et al., 2005), robust safety in our system would be a consequence of the typing rules for the opponent, and the scoping rules underlying groups.

Other desirable extensions for the system include the ability to express the revocation of capabilities, to change the ownership on resources, and to accommodate some form of structure (e.g. partial order) in the domain of principals. For instance, in our example of the on-line editorial system, one could envisage a notion of substitutivity, so that a single rôle, say Staff, could be used to represent both Journal and Editor in all situations in which this may be useful.

References

Abadi, M. and Gordon, A. D. (1997). Reasoning about cryptographic protocols in the Spi calculus. In *CONCUR'97*, volume 1243 of *LNCS*, pages 59–73. Springer-Verlag.

Brandt, M. and Henglein, F. (1998). Coinductive axiomatization of recursive type equality and subtyping. *Fundaenta Informaticae*, 33(4):309–338.

Bugliesi, M., Colazzo, D., and Crafa, S. (2004). Type based Discretionary Access Control. In *CONCUR'04*, volume 3170 of *LNCS*, pages 225–239. Springer-Verlag.

Bugliesi, M. and Giunti, M. (2007). Secure implementations of typed channel abstractions. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 251–262. ACM.

Carbone, M., Honda, K., and Yoshida, N. (2007). Structured communication-centred programming for web services. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Braga, Portugal, March 24 - April 1*, Lecture Notes in Computer Science, pages 2–17. Springer.

Cardelli, L., Ghelli, G., and Gordon, A. D. (2005). Secrecy and group creation. *Information and Computation*, 196(2):127–155.

Chothia, T., Duggan, D., and Vitek, J. (2003). Type-based Distributed Access Control. In *CSFW'03*, pages 170–184. IEEE Computer Society.

Fournet, C., Gordon, A., and Maffeis, S. (2007). A type discipline for authorization in distributed systems. In *CSF'07*, pages 31–48. IEEE Computer Society.

Gapeyev, V., Levin, M. Y., and Pierce, B. C. (2002). Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548.

Gordon, A. and Jeffrey, A. (2004). Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3–4):435–483.

Gorla, D. and Pugliese, R. (2003). Resource access and mobility control with dynamic privileges acquisition. In *ICALP'03*, LNCS, pages 119–132. Springer.

Hennessy, M., Rathke, J., and Yoshida, N. (2005). SafeDpi: a language for controlling mobile code. *Acta Informatica*, 42(4–5):227–290.

Hennessy, M. and Riely, J. (2002a). Information flow vs resource access in the asynchronous π-calculus. *ACM Transactions on Programming Languages and Systems*, 24(5):566–591.

Hennessy, M. and Riely, J. (2002b). Resource access control in systems of mobile agents. *Information and Computation*, 173:82–120.

Honda, K., Vasconcelos, V., and Kubo, M. (1998). Language primitives and type discipline for structured communication-based programming. In *ESOP '98*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag.

Honda, K., Vasconcelos, V., and Yoshida, N. (2000). Secure information flow as typed process behaviour. In *ESOP '00*, volume 1782 of *LNCS*, pages 188–199. Springer-Verlag.

Honda, K., Yoshida, N., and Carbone, M. (2008). Multiparty asynchronous session types. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12*, pages 273–284. ACM.

Kobayashi, N. (2005). Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4):291–347.

Lampson, B. (1974). Protection. *ACM Operating Systems Rev.*, 8(1):18–24.

McCollum, C., Messing, J. R., and Notargiacomo, L. (1990). Beyond the pale of MAC and DAC – defining new forms of access control. In *Proc. of IEEE Symposium on Security and Privacy*, pages 190–200.

Myers, A. C. and Liskov, B. (2000). Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol*, 9(4):410–442.

Nicola, R. D., Ferrari, G. L., Pugliese, R., and Venneri, B. (2000). Types for access control. *Theor. Comput. Sci.*, 240(1):215–254.

Nicola, R. D., Gorla, D., and Pugliese, R. (2006). Confining data and processes in global computing applications. *Sci. Comput. Program.*, 63(1):57–87.

Pierce, B. and Sangiorgi, D. (1996). Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5).

Pottier, F. (2002). A simple view of type-secure information flow in the π-calculus. In *CSFW'02*, pages 320–330. IEEE Computer Society.

Riely, J. and Hennessy, M. (1998). A typed language for distributed mobile processes (extended abstract). In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January, San Diego, CA, USA*, pages 378–390. ACM Press.

Samarati, P. and di Vimercati, S. D. C. (2001). Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design*, volume 2171 of *LNCS*. Springer-Verlag.

Sandhu, R. S. and Munawer, Q. (1998). How to do discretionary access control using roles. In *ACM Workshop on Role-Based Access Control*, pages 47–54.

Schneider, F. B. (2000). Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50.

Sewell, P. and Vitek, J. (2003). Secure composition of untrusted code: Box π, wrappers and causality types. *Journal of Computer Security*, 11(2):135–188.